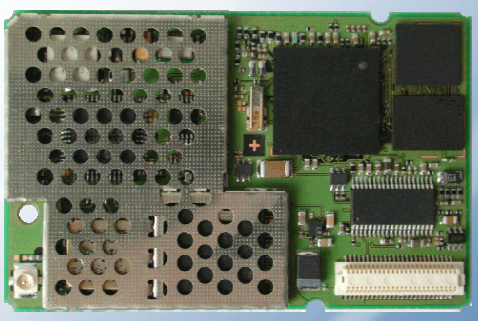
SIEMENS

TC45

Siemens Cellular Engine



JAVA™ User's Guide

Version: 02
DocID: TC45_JAVA User's Guide_V02

Wireless Modules

Document Name:	**TC45 JAVA User's Guide**
Version:	**02**
Date:	**June 30, 2003**
DocId:	**TC45_JAVA User's Guide_V02**
Status:	**Confidential / Released**

General Notes
Product is deemed accepted by recipient and is provided without interface to recipient's products. The documentation and/or product are provided for testing, evaluation, integration and information purposes. The documentation and/or product are provided on an "as is" basis only and may contain deficiencies or inadequacies. The documentation and/or product are provided without warranty of any kind, express or implied. To the maximum extent permitted by applicable law, Siemens further disclaims all warranties, including without limitation any implied warranties of merchantability, completeness, fitness for a particular purpose and non-infringement of third-party rights. The entire risk arising out of the use or performance of the product and documentation remains with recipient. This product is not intended for use in life support appliances, devices or systems where a malfunction of the product can reasonably be expected to result in personal injury. Applications incorporating the described product must be designed to be in accordance with the technical specifications provided in these guidelines. Failure to comply with any of the required procedures can result in malfunctions or serious discrepancies in results. Furthermore, all safety instructions regarding the use of mobile technical systems, including GSM products, which also apply to cellular phones must be followed. Siemens or its suppliers shall, regardless of any legal theory upon which the claim is based, not be liable for any consequential, incidental, direct, indirect, punitive or other damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information or data, or other pecuniary loss) arising out the use of or inability to use the documentation and/or product, even if Siemens has been advised of the possibility of such damages. The foregoing limitations of liability shall not apply in case of mandatory liability, e.g. under the German Product Liability Act, in case of intent, gross negligence, injury of life, body or health, or breach of a condition which goes to the root of the contract. However, claims for damages arising from a breach of a condition, which goes to the root of the contract, shall be limited to the foreseeable damage, which is intrinsic to the contract, unless caused by intent or gross negligence or based on liability for injury of life, body or health. The above provision does not imply a change on the burden of proof to the detriment of the recipient. Subject to change without notice at any time. The interpretation of this general note shall be governed and construed according to German law without reference to any other substantive law.

Copyright
Transmittal, reproduction, dissemination and/or editing of this document as well as utilization of its contents and communication thereof to others without express authorization are prohibited. Offenders will be held liable for payment of damages. All rights created by patent grant or registration of a utility model or design patent are reserved.

Copyright © Siemens AG 2003

Trademark notices
MS Windows® is a registered trademark of Microsoft Corporation.
Java™ and Sun™ ONE Studio 4.1 Mobile Edition are registered trademarks of Sun Microsystems Inc.
Borland® JBuilder® Enterprise MobileSet 3 and JBuilder® Personal MobileSet 3 are registered trademarks of Borland Software Corporation

Table of Contents

0 Document history 9

1 Introduction 10

1.1 Overview 10

1.2 Related documents 10

1.3 Terms and abbreviations 11

2 Installation 12

2.1 System requirements 12

2.2 Installation CD 12

2.2.1 Components 13
2.2.1.1 WinMUX 2.1 13
2.2.1.2 Module Exchange Suite 13
2.2.1.3 IMPSiemens 13
2.2.1.4 AT Java Open Framework (AJOF) 13

2.3 SMTK installation 14

2.3.1 Installing the Standard Development Toolkit 14
2.3.2 Installing the SMTK environment 14
2.3.3 Installing Sun ONE Studio 4.1 Mobile Edition 15
2.3.4 Installing Borland JBuilder MobileSet 3 15
2.3.4.1 JBuilder Enterprise Edition MobileSet 3 15
2.3.4.2 JBuilder Personal Edition MobileSet 3 15

2.4 Uninstall 16

2.4.1 Uninstall WinMux 16

2.5 Upgrades 16

3 Maintenance 17

3.1 Power Saving 17

3.2 Charging 17

3.3 Alarm mode 18

3.4 Shut down 18
3.4.1 Automatic shutdown 18
3.4.2 Restart after switch off 18

3.5 Special AT command set for Java applications 19
3.5.1 Switching from data mode to command mode 19
3.5.2 Mode indication after MIDlet startup 19
3.5.3 Long responses 19

3.6 Restrictions 20
3.6.1 Flash file system 20
3.6.2 Memory 20

3.7 Performance statements 20
3.7.1 Java 20
3.7.2 Pin-IO 21

4 Software platform 22

4.1 Software architecture 22

4.2 Interfaces 23
4.2.1 ASC0 - Serial device 23
4.2.2 General Purpose I/O 23
4.2.3 ASC1 23
4.2.4 Digital Audio Interface (DAI) 23
4.2.5 JVM interfaces 24
4.2.5.1 IMP 1.0 API to TCP/IP 24

4.3 Data flow of a Java application running on the module 25

4.4 Handling interfaces and data service resources 26
4.4.1 Module states 26
4.4.1.1 State 1: default – no Java application is running 27
4.4.1.2 State 2: no Java application is running, general purpose I/O 27
4.4.1.3 State 3: no Java application is running, AT interface on ASC1 27
4.4.1.4 State 4: default – Java application active 27
4.4.1.5 State 5: Java application active, general purpose I/O 28
4.4.1.6 State 6: Java application active, Java interface to ASC1 28
4.4.2 Module state transitions 29

5 MIDlets 31

5.1 MIDlet documentation 31

5.2 MIDlet life cycle 31

5.3 Hello World MIDlet 32

6 AT Java Open Framework (AJOF) 34

6.1 AT Commands 34
6.1.1 Mobile engine status 34
6.1.2 Voice call handling 34
6.1.3 CSD call and GPRS connection handling 34
6.1.4 Short Message Service 34
6.1.5 Phonebook features 35
6.1.6 Pin I/O 35

7 File transfer to module 36

7.1 Module Exchange Suite 36
7.1.1 Windows based 36
7.1.2 Command line based 36

7.2 Over the Air Provisioning 36

7.3 Security issues 36
7.3.1 Module Exchange Suite 37
7.3.2 OTAP 37

8 Over The Air Provisioning (OTAP) 38

8.1 Introduction to OTAP 38

8.2 OTAP overview 38

8.3 OTAP Parameters 39

8.4 Short message format 40

8.5 Java file format 41

8.6 Procedures 42
8.6.1 Install/Update 42
8.6.2 Delete 43

8.7 Time out values and result codes 44

8.8 Notes 44

8.9 How to 45

9 Compiling and running a program without a Java IDE 46

9.1 Build results 46
9.1.1 JAD file 46

9.2 Compile and run on a PC 47

9.3 Run on the module with manual start 47

9.4 Run on the module with Autostart 47
9.4.1 Switch on Autostart 48
9.4.2 Switch off Autostart 48

10 Debug environment 49

10.1 Data flow of a Java application in the debug environment 49
10.1.1 Serial port and pin I/O handling differences 50
10.1.2 Restrictions 50
10.1.3 RS-232 emulator 50

10.2 Java IDE 50
10.2.1 Sun ONE Studio 4.1 Mobile Edition 51
10.2.1.1 Switching emulators 52
10.2.1.2 Projects 53
10.2.1.3 Templates 54
10.2.1.4 Examples 54
10.2.1.5 Compile and run 54
10.2.2 JBuilder Mobile Edition 54
10.2.2.1 TC45 SMTK in Personal Edition 54
10.2.2.1.1 Integration of TC45 SMTK 55
10.2.2.1.2 Restoring original settings 57
10.2.2.2 TC45 SMTK in Enterprise Edition 58
10.2.2.3 Examples 59
10.2.2.4 Projects and templates 60
10.2.2.5 Compile and run 61

10.3 Breakpoints 61

11 Java Tutorial 62

11.1 Using the AT command API 62
11.1.1 Class ATCommand 62
11.1.1.1 Instantiation with or without CSD support 62
11.1.1.2 Sending an AT command to the device, the *send()* method 63
11.1.1.3 Data connections 63
11.1.1.4 Synchronization 64
11.1.2 ATCommandResponseListener interface 65
11.1.2.1 Non-blocking *ATCommand.send()* method 65
11.1.3 ATCommandListener interface 65

11.1.3.1 ATEvents....66
11.1.3.2 Implementation....66
11.1.3.3 Registering a listener with an *ATCommand* instance....67

11.2 Programming the MIDlet....68
11.2.1 Threads....68
11.2.2 Example....68

11.3 AJOF....70
11.3.1 Example....70

Figures

Figure 1: Overview 10
Figure 2: Test overview 21
Figure 3: Interface Configuration 24
Figure 4: Data flow of a Java application running on the module 25
Figure 5: Module State 1 27
Figure 6: Module State 2 27
Figure 7: Module State 3 27
Figure 8: Module State 4 27
Figure 9: Module State 5 28
Figure 10: Module State 6 28
Figure 11: Module State Transition Diagram 29
Figure 12: OTAP Overview 38
Figure 13: OTAP: Install/Update Information Flow The messages in brackets are optional 42
Figure 14: OTAP: Delete Information Flow 43
Figure 15: Data flow of a Java application in the debug environment 49
Figure 16: Sun ONE Studio - The installed emulators. 51
Figure 17: Sun ONE Studio - Switching to the Siemens emulator. 52
Figure 18: Sun ONE Studio - Project Manager 53
Figure 19: Sun ONE Studio - Selecting a template 54
Figure 20: Different JDK must be set in User Home 55
Figure 21: Dialog box after changing the JDK to Siemens IMP Wireless Toolkit. 56
Figure 22: Original setting for User Home 57
Figure 23: JBuilder - JDK settings 58
Figure 24: JBuilder - Siemens Library 58
Figure 25: JBuilder - Sample Projects 59
Figure 26: JBuilder - Configuring a new project. 60
Figure 27: JBuilder - Project Properties 61

Tables

Table 1: Java AT commands 30
Table 2: A typical sequence of MIDlet execution is: 32
Table 3: Parameters and keywords 39

0 Document history

Preceding document: "TC45 JAVA User's Guide" Version 01
New document: "TC45 JAVA User's Guide" Version **02**

Chapter	Page	What is new
11.3.1	70	Revised example

1 Introduction

1.1 Overview

The TC45 module features an ultra-low profile and low-power consumption for data, voice, SMS and fax. With Java technology and up to nine digital I/O ports, the module enables easy integration of your application.
This document explains how to work with the TC45 module, the installation CD and the tools provided on the installation CD.

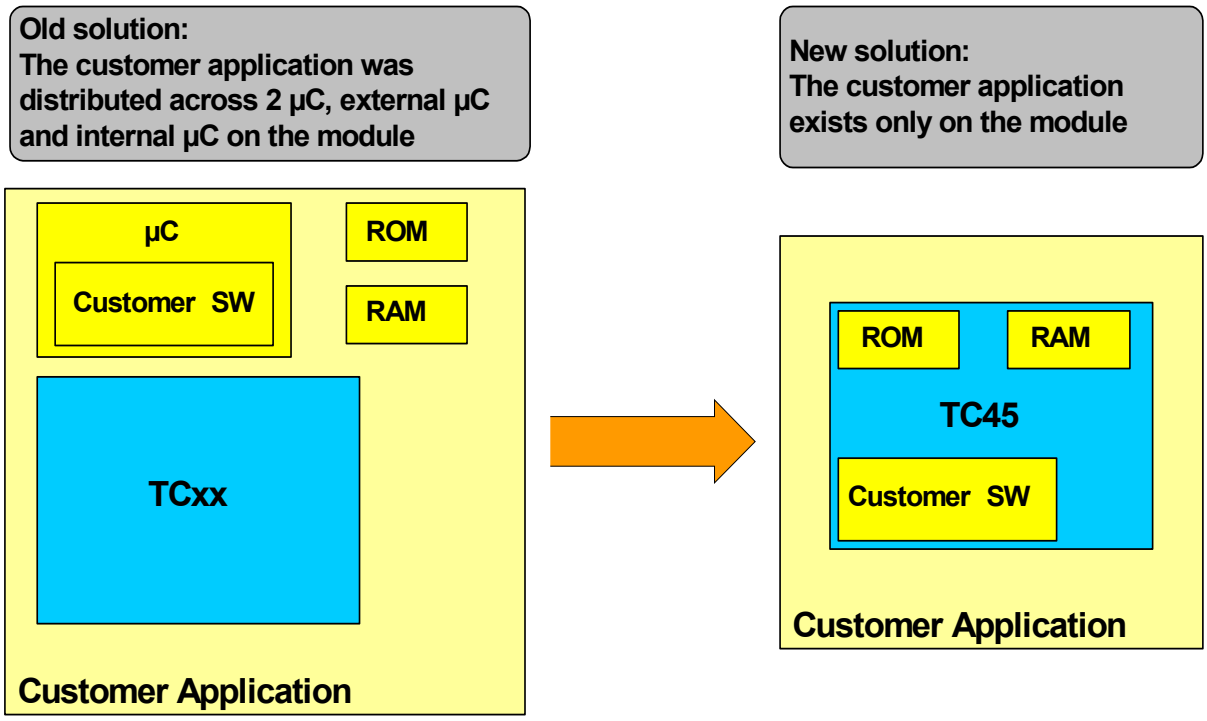


Figure 1: Overview

1.2 Related documents

In addition to the Java Docs for the development API (see Chapter 4), the following documents are included with the SMTK:

[1] Multiplexer Installation Guide
[2] Application Note 23: Installing TC45 Module on DSB45
[3] DSB45 Support Box - Evaluation Kit for Siemens Cellular Engines
[4] TC45 AT Command Set
[5] TC45 Hardware Interface Description
[6] Java doc \IMPSIEMENS\doc\html\index.html

1.3 Terms and abbreviations

Abbreviation	Description
API	Application Program Interface
ASC	Asynchronous Serial Connection
CLDC	Connected Limited Device Configuration
CSD	Circuit-Switched Data
DAI	Digital Audio Interface
DCD	Data Carrier Detect
DSR	Data Set Ready
GPIO	General Purpose I/O
GPRS	General Packet Radio Service
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
IMP	Information Module Profile
IP	Internet Protocol
J2ME™	Java 2 Mobile Edition
J2SE™	Java 2 Standard Edition
JAD	Java Application Description
JAR	Java Archive
JDK	Java Development Kit
JVM	Java Virtual Machine
LED	Light Emitting Diode
ME	Mobile Engine
MIDP	Mobile Information Device Protocol
OTA	Over The Air
OTAP	Over The Air Provisioning of Java Applications
PDP	Packet Data Protocol
PDU	Protocol Data Unit
SDK	Standard Development Kit
SMS	Short Message Service
SMTK	Siemens Mobile Toolkit
TCP	Transfer Control Protocol
URC	Unsolicited Result Code
URL	Universal Resource Locator
VBS	Visual Basic Script

2 Installation

2.1 System requirements

The Siemens Mobility Toolkit (SMTK) TC45 requires that you have:

1. Windows 2000 or Windows XP installed
2. 35 Mbytes free disk space
3. Administration privileges
4. Java 2 SDK, Standard Edition version 1.3 or 1.4. To install the JDK version 1.4.1 provided, follow the instructions in Section 2.3.1. In case you wish to use the Java 2 SDK version 1.3 please download the version from http://java.sun.com/j2se/1.3/download.html

If a Java IDE such as Sun ONE Studio 4.1 Mobile Edition, Borland JBuilder Enterprise MobileSet 3 or JBuilder Personal MobileSet 3 is installed, it can be integrated into the SMTK environment during the installation of the SMTK. To install Sun ONE Studio or Borland JBuilder, follow the installation instructions in Section 2.3.3 and Section 2.3.4 respectively.

2.2 Installation CD

The Siemens Mobility Toolkit TC45 Installation CD includes:

- MuxInstall.pdf
  - mux_driver_installation_guide_vxx.pdf
- Module Exchange Suite
- IMPSiemens
  - bin
  - classes
  - doc
    - html
      - javadocs for STMK APIs
  - lib
    - classes.zip
  - src
    - diverse examples
- AT Java Open Framework
- JDK 1.4.1_02
  - J2sdk-1_4_1_02-windows-i586.exe
- Sun One Mobile Edition
  - ffj_me_win32.exe
- Documents:
  - dsb_45_an23_inst_tc45_vxx.pdf
  - dsb45_hd_vxx.pdf
  - tc45_atc_vxx.pdf
  - tc45_hd_vxx.pdf
- java_user_guide_v01.pdf (this document)

Please note that “vxx” refers to the version number of the document.

2.2.1 Components

2.2.1.1 WinMUX 2.1

The multiplexer mode enables one serial interface to transmit data to three different customer applications. This is achieved by providing three virtual channels using a multiplexer. The multiplexer is necessary for the SMTK environment. Information how to install the provided multiplexer sources can be found in [1].

2.2.1.2 Module Exchange Suite

The Module Exchange Suite allows the developer to access the Flash file system on the cellular engine from the development environment over a serial interface. File transfers from PC to module are greatly facilitated by this suite.

2.2.1.3 IMPSiemens

IMPSiemens is the directory where all the necessary components for the SMTK are stored. The virtual machine, all necessary interfaces and packages, the RS-232 emulator and the Flash file system emulator are included. The AT Command API is the Java interface to the module's AT command interpreters. It allows AT commands to be sent, as strings, directly to the AT command interpreters.

2.2.1.4 AT Java Open Framework (AJOF)

The AJOF sits on top of the AT Command class API and allows the developer to write Java applications for the module without dealing directly with AT Commands.

2.3 SMTK installation

The SMTK comes with an installation CD. The installation program provided automatically installs the necessary interfaces and emulators. Software can be uninstalled and updated with the install program. The next sections cover the installation and removal of the SMTK and the installation of the SDK, Sun ONE Studio or Borland JBuilder.

2.3.1 Installing the Standard Development Toolkit

1. The JDK version 1.4.1_02 is provided on the TC45 SMTK installation disk in the subdirectory “JDK 1.4.1”. To begin the installation, start the j2sdk-1_4_1_02-windows-i586.exe and follow the instructions of the JDK setup procedure.
2. Once the toolkit has been installed, the environment variable “path” can be altered to comfortably use the JDK tools.
3. Open the Control Panel.
   a) Open *System*.
   b) Click on *Advanced*.
   c) Click on the *Environment Variables* button.
   d) Choose *path* from the list of system variables.
   e) Append the path for the bin directory of the newly installed SDK to the list of directories for the *path* variable

2.3.2 Installing the SMTK environment

1. Insert CD, start setup.exe. If the dialog box appears simply press the “Next” button to continue the procedure.
2. You will be asked to read the license agreement. If you accept the agreement, press “Next” to continue with the installation.
3. A file including special information about the installation and use of the SMTK is shown. Press “Next” to continue.
4. The installation software checks for the SDK. If there is no SDK on the system the installation procedure will not continue. Please follow the instructions in Section 2.3.1 above for installing the SDK.
5. At this point, the installation software checks for a Java IDE to be integrated with the SMTK. A Java IDE is not necessary to use the TC45 SMTK. The IDE installation can be done at any time even if the TC45 SMTK is installed already. To install the Java IDE run the setup program in maintenance mode again. However, you can continue the setup procedure and install the IDE installation later or cancel the setup program at this stage and restart it after installing one of the supported Java IDEs. In case you wish to install a Java IDE please follow the instructions in Section 2.3.3 for installing Sun ONE Studio Mobile Edition, Section 2.3.4 for installing Borland JBuilder Enterprise Mobile Set 3 or Section 2.3.4.2 for installing Borland JBuilder Personal Mobile Set 3.
6. If the SDK and one or more Java IDE’s are found, you will be asked to choose which IDE you want integrated into the TC45 development environment. Once an IDE has been found and selected, press next to continue. Ensure that your Java IDE is closed.
7. Select the folder where the TC45 SMTK will be installed. A folder will be suggested to you but you may browse to select a different one.
8. Choose the path that TC45 will appear under in the Start Menu.
9. A brief summary of all entries made shows up.
10. After step 9, all necessary files will be copied from the CD into the target folder.

11. The installation of WinMux 2.1 is started.
    a) Ensure that the module is connected to a serial port and turn the module power on.
    b) Press the “Scan” button of the application. All Siemens modems found will be listed in a box. The virtual ports are shown. If at least one modem is found, the “Install” button becomes active. The selected modems are installed if this button is pressed.
    c) The Device Manager can be used to check if the installation was successful. The driver instances are visible in the device manager class “Multi-port serial Adapters”.
12. This is the final step. Again, a listing of all installed components appears.

2.3.3 Installing Sun ONE Studio 4.1 Mobile Edition

1. Sun ONE Studio 4.1 Mobile Edition is provided on the TC45 SMTK installation disk in the subdirectory “SunOne ME” To begin installation, start the ffj_me_win32.exe and follow the instructions of the Sun ONE setup procedure.
2. On the first use of Sun™ ONE Studio 4 Update 1 Mobile Edition after installation, you will be prompted to specify a personal Java folder when Sun ONE is started for the first time. If more then one user uses the computer, each user may have their own Java folder.

   Note: The integration of the SMTK into Sun™ ONE Studio 4 Update 1 Mobile Edition is only possible if the personal user folder is set. It can only be rolled back by the user who installed the SMTK. If all users use the same Java folder, any user may roll back the integration.

2.3.4 Installing Borland JBuilder MobileSet 3

Borland JBuilder and MobileSet can be purchased from http://www.borland.com/jbuilder. JBuilder must be installed before installing the MobileSet. MobileSet 3 for Personal Edition is the same as for Enterprise Edition.

Note: The installation path name of JBuilder should not include space characters. Otherwise MobileSet is not working.

2.3.4.1 JBuilder Enterprise Edition MobileSet 3

A free version of JBuilder Enterprise Edition is available for 30 days. JBuilder must be installed before installing the MobileSet. Installation instructions can also be found on the web page.

2.3.4.2 JBuilder Personal Edition MobileSet 3

A free version for non-commercial use can be downloaded from Borland’s website. Installation is similar to Enterprise Edition. JBuilder must be installed before installing the MobileSet. The integration of TC45 SMTK into the JBuilder Personal Edition is somewhat tricky and therefore briefly explained chapter 10.2.2.1 TC45 SMTK in Personal Edition.

2.4 Uninstall

The TC45 SMTK install package comes with an uninstall facility. The entire SMTK or parts of the package can be removed. To start the uninstall facility, open the *Control Panel*, select *Add/Remove Programs*, select *TC45* and follow the instructions.

2.4.1 Uninstall WinMux

The WinMux multiplexer must be manually uninstalled with the Hardware Manager.
It can also be uninstalled with *Control Panel->Add/Remove Hardware*. Choose the uninstall options and select the *Serial Multiplexer* to be uninstalled.

2.5 Upgrades

The SMTK can be modified, repaired or removed by running the setup program on the Installation CD.

3 Maintenance

Basic maintenance features of the TC45 are described below. Explicit details of these functions and modes can be found in the AT Command Set [4] and the Hardware Interface Description [5].

3.1 Power Saving

The module supports several power saving modes which can be configured by the at command *at+cfun* [4]. Power saving affects the Java application in two ways. On the one hand it limits the access to the serial interface (RS-232-API) and the GPIO pins and on the other hand power saving efficiency is directly influenced by the way a Java application is programmed.

Java hardware access limitations:

- In NON-CYCLIC SLEEP mode (cfun=0) the serial interface cannot be accessed while in CYCLIC SLEEP mode (CFUN=5, 6, 7 or 8) the serial interface can be used with hardware flow control (CTS/RTS).
- In all SLEEP modes the GPIO polling frequency is reduced, so that only signal changes which are less than 0.2Hz can be detected properly. Furthermore it should be mentioned that the signal must be constant for at least 2.12s to detect changes. For further details refer to [5].

Java power saving efficiency:

- As long as any Java thread is active, power consumption cannot be reduced, regardless whether any SLEEP mode has been activated or not. So a Java application that wants to be power efficient should not have any unnecessarily active threads (e.g. no busy loops).

3.2 Charging

Please refer to [4] and [5] for general information about charging. Charging can be monitored by the running Java application. The JVM is active in Charge mode and when autostart is activated also in Charge-Only mode. Only a limited number of AT commands are available when the module is in Charge-Only mode. A Java application must be able to handle the Charge-Only mode and reset the module to reinstate the normal mode. See [5] for information about the Charge-Only mode.

The Charge-Only mode is indicated by URC “SYSSTART CHARGE-ONLY MODE”.

Note: When a Java application is started in Charge-Only mode only AT Command API’s without CSD are available. The indicating URC is created after issuing the very first AT command on any opened channel. To read the URC it is necessary to register a listener (see [6]) on this AT command API instance before passing the first AT command.

3.3 Alarm mode

The ALARM can be set by the *at+cala* AT command. Please refer to the AT Command Set [4] and Hardware Interface Description [5] for more information. One can set an alarm, switch off the module with *at^smso*, and have the module restart in ALARM mode at the time set with *at+cala*. Only a limited number of AT commands are available when the module is in ALARM mode. The JVM is active in ALARM mode when autostart is activated. A Java application must be able to handle the ALARM mode and reset the module to reinstate the normal mode.
The ALARM mode is indicated by URC “SYSSTART ALARM MODE”.

Note: When a Java application is started in ALARM mode only AT Command API’s without CSD are available. The indicating URC is created after issuing the very first AT command on any opened channel.

3.4 Shut down

In the case of an unexpected shut down, data that should be written will get lost due to a buffered write access to the flash file system. However, the best and safest approach to powering down the module is to issue the AT^SMSO command. This procedure lets the engine log off from the network and allows the software to enter into a secure state and save all data. Further details can be found in [5].

3.4.1 Automatic shutdown

The module is switched off automatically in different situations:

- under- or overtemperature
- under- or overvoltage

This will happen without a warning notification unless the appropriate URC has been activated. If the URC is activated, the module will deliver the URC before switching off. The AT command *at^sctm* is used to activate and configure the overheating URC. It is recommended that this URC be activated so that the module can be shut by the application with *at^smso* after setting an alarm, see Section 3.3. Please note that there is no URC function available for overvoltage conditions, i.e. no alert will be sent before shutdown. The conmands are described in the AT Command Set [4] while a description of the shutdown procedure can be found in [5].

3.4.2 Restart after switch off

The module can be switched off with the AT command, *at^smso* without setting an alarm time, see the AT Command Set [4]. A power failure will also switch off the module. When the module is switched off, external hardware must restart the module with the Ignition line (IGT). The Hardware Interface Description [5] explains how to handle a switched off situation.

3.5 Special AT command set for Java applications

For the full AT command set refer to [4]. There are differences in the behaviour of issuing AT commands from a Java application compared to using AT commands over a serial interface.

3.5.1 Switching from data mode to command mode

Cancelling the data flow with “+++” is not available in Java applications, see [4] for details. To break the data flow use *breakConnection(),* refer to \IMPSIEMENS\doc\index.html [6].

3.5.2 Mode indication after MIDlet startup

As on the serial interface after starting the module without autobauding on, the module sends its state (^SYSSTART, ^SYSSTART ALARM MODE etc.) to the MIDlet. This is done via URC to the AT Command API instance which executes the very first AT Command from within Java. To read this URC it is necessary to register a listener (see [8]) on this AT Command API instance before passing the first AT Command.

3.5.3 Long responses

The AT Command API can handle responses of AT commands up to a length of 1024 bytes. Some AT commands have responses longer than 1024 bytes, for these responses the Java application will receive an Exception.

Existing workarounds:

- Instead of listing the whole phone book, read the entries one by one
- Instead of listing the entire short message memory, again list message after message
- Similarly, read the provider list piecewise
- Periods of monitoring commands have to be handled by Java, i.e. *at^moni*, *at^smong*. These AT commands have to be used without parameters, i.e. *at^moni* the periods have to be implemented in Java.

3.6 Restrictions

3.6.1 Flash file system

The maximum length of a complete path name, including the path and the filename, is limited by the Flash file system on the module to 124 characters.

3.6.2 Memory

The memory model of the modules operating system uses 16kByte memory pages. Due to this architecture there are two restrictions:

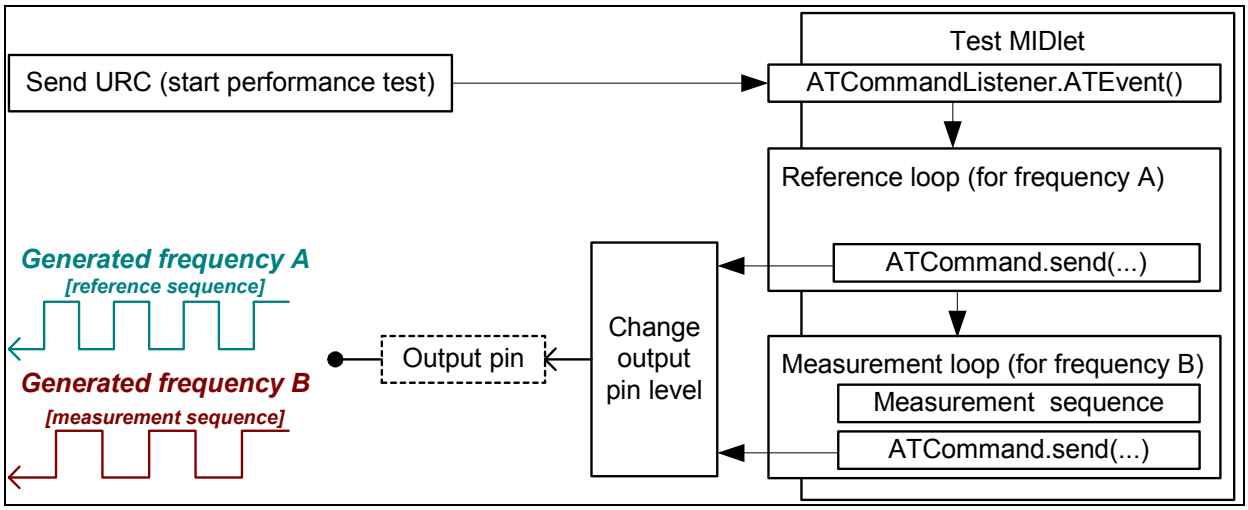
- It is not possible to create an object that needs more than 16 Kbytes linear (i.e. non-fragmented) memory. If an object has to be created whose size exceeds these 16kByte an OutOfMemoryError will be thrown.
- If the memory is unfavourably fragmented it might not be possible to create an object that needs a larger amount of memory (smaller than 16kBytes) even though the reported amount of free memory is larger than the amount needed by the object. If this happens an OutOfMemoryError will be thrown.

3.7 Performance statements

Scope of the performance study was getting comparable values that indicate the performance under certain circumstances.

3.7.1 Java

This section gives information about the Java command execution throughput (“jPS”= Java statements per second). The scope of this measurement is only the statement execution time, not the execution delay (Java command on AT interface ➔ Java instruction execution ➔ reaction on GPIO).



The following Java instruction was used for calculation of the typical jPS:

value = (2 x number of calculation statements) /
((1 / frequencyB) - (1 / frequencyA));

Measurement and calculation were done using:

- **duration of each loop** = 600 s
- **number of calculation statements** = 5 "**result=(CONSTANT_VALUE/variable_value);**"-Instructions
  (executed twice per pin cycle)
- **frequencyA** as measured with universal counter
- **frequencyB** as measured with universal counter

The reference loop has the same structure as the measurement loop except the measurement sequence is moved.

State	jPS-Value (mean)
TC45 module in IDLE mode / Not connected	~750 jPS
CSD connection	~450 jPS

These mean values may be sporadically reduced depending on dynamic conditions.

3.7.2 Pin-IO

The pin IO test was defined to find out how fast a Java MIDlet can process URCs caused by Pin IO and react on these URCs.
The URCs are generated by feeding an input pin with an external frequency. As soon as the Java MIDlet gets informed about the URC, it tries to regenerate the feeding frequency by toggling another output pin.

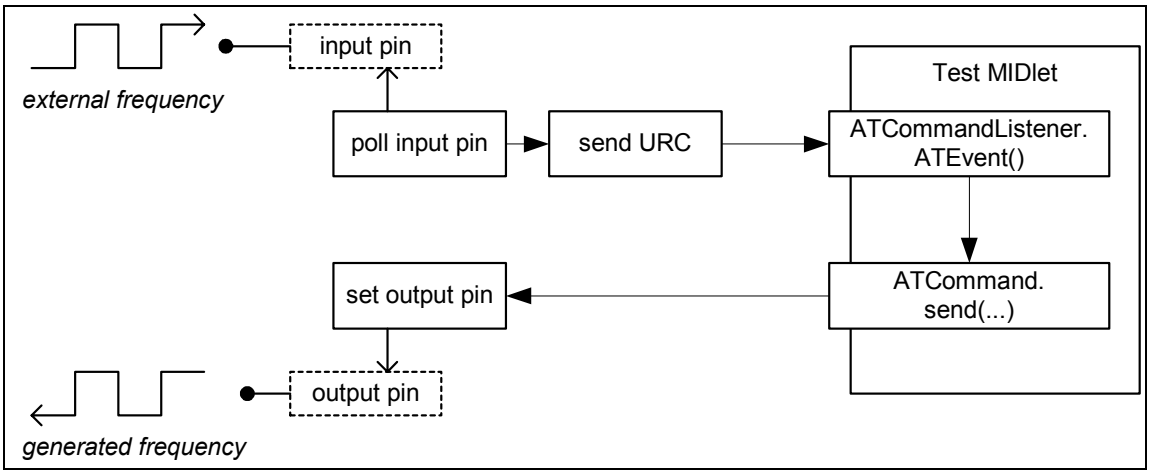


Figure 2: Test overview

The results of this test show that the delay from changing the state on the pin to processing the URC in the MIDlet is at least 20 TDMA frames, but depends mainly on the amount of garbage to collect and number of thread to serve by the virtual machine. So Pin IO is not suitable to generate or detect frequencies.

4 Software platform

In this chapter, we discuss the software architecture of the SMTK and the interfaces to it.

4.1 Software architecture

The SMTK enables a customer to develop a Java application on a PC and have it be executable on the TC45 module. The application is loaded onto the module. The SMTK comprises:

- A virtual machine: the Java™ 2 Micro Edition (J2ME™)
  The J2ME™ is provided by SUN Microsystems, http://java.sun.com/j2me/. It is specifically designed for embedded systems and has a small memory footprint.
- IMP 1.0
  The IMP is based on the standard MIDP 1.0 and does not include the graphical interface LCDUI.
  TCP/IP API is part of IMP.

- Additional Java virtual machine interfaces:
  AT Command API
  RS-232 API
  File I/O API

  The data flow through these interfaces is shown in Figure 4 and Figure 15.

  Note: RS-232 API does not presume that the serial interface is using the RS-232 signal level, refer to Hardware Interface Description [5] about signal levels.

- Memory space for Java programs:
  Flash File System: around 300k
  RAM: around 100k

  Programs and data share the space in the Flash file system and in RAM.

- Additional accessible periphery for java applications
  - A maximum of 9 shared digital I/O pins usable, for example, as:
    - Output: status LEDs
    - Input: Emergency Button
  - Serial interface (RS-232 API): This standard serial interface could be used, for example, with an external GPS device or a current meter. Detailed information are available in Chapter 4.2.

4.2 Interfaces

4.2.1 ASC0 - Serial device

ASC0, an Asynchronous Serial Connection, is a 9-wire serial interface. It is described in the Hardware Interface Description [5]. Without a running Java application the module can be controlled by sending AT commands via ASC0. Furthermore ASC0 is designed for transferring files from the development PC to the module and for controlling the module with AT commands. When a Java application started, ASC0 is used for debugging from the development environment, as System.out for Java applications running on the module (see Section 4.4.1).

4.2.2 General Purpose I/O

There are nine I/O pins that can be configured for general purpose I/O. When JAVA User's Guide starts up, all 9 pins of both interfaces are set, by default, to high-impedance state for use as input.
The pins are divided into two GPIO pin groups. Pin group 0, (GPIO 0 – 3), is the set also used as ASC1. The second pin group, (GPIO 4 – 8), shares resources with the DAI. Pins 0 - 3 can be configured either as general purpose I/O (GPIO) or ASC1 but not both. The same for pins 4 - 8, they can be configured either as GPIO or as DAI. See [4] and [5] about configuring the pins.

4.2.3 ASC1

ASC1 is the second serial interface on the module. This is a 4-pin interface (RX, TX, RTS, CTS). It can be used as a second AT interface when a Java application is not running or by a running Java application as an RS-232 port, refer to Java doc [6]. These four pins can be configured as general purpose I/O when not used as ASC1. The default status of ASC1 is inactive.

4.2.4 Digital Audio Interface (DAI)

To support the DAI function, the TC45 has a five-line serial interface with one input data clock line and input/output data and frame lines. These five pins can also be configured as general purpose I/O when not used as DAI. Refer to AT Command Set [4] and Hardware Interface Description document [5] for more information.

4.2.5 JVM interfaces

IMP 1.0	File API	RS232 API	AT Command API
Connected Limited Device Configuration (CLDC)			
J2ME			

Figure 3: Interface Configuration

J2ME, CLDC and MIDP were implemented by SUN. IMP 1.0 is a stripped down version of MIDP prepared by Siemens and does not include the graphical interface LCDUI. Siemens developed the File I/O API and the AT command API. TC45 also provides an RS-232 interface related to the RS-232 API of MIDP2.0. Documentation for J2ME and CLDC can be found at http://java.sun.com/j2me/. Documentation for the other APIs is found in …/ Java doc [6].

4.2.5.1 IMP 1.0 API to TCP/IP

IMP 1.0 provides access to TCP/IP with these interfaces:

- DatagramConnection
- HttpConnection
- StreamConnection

Because the used network connection, CSD or GPRS, is fully transparent to the Java interface, the CSD and GPRS parameters must be defined separately by the AT command at^sjnet [4].

4.3 Data flow of a Java application running on the module

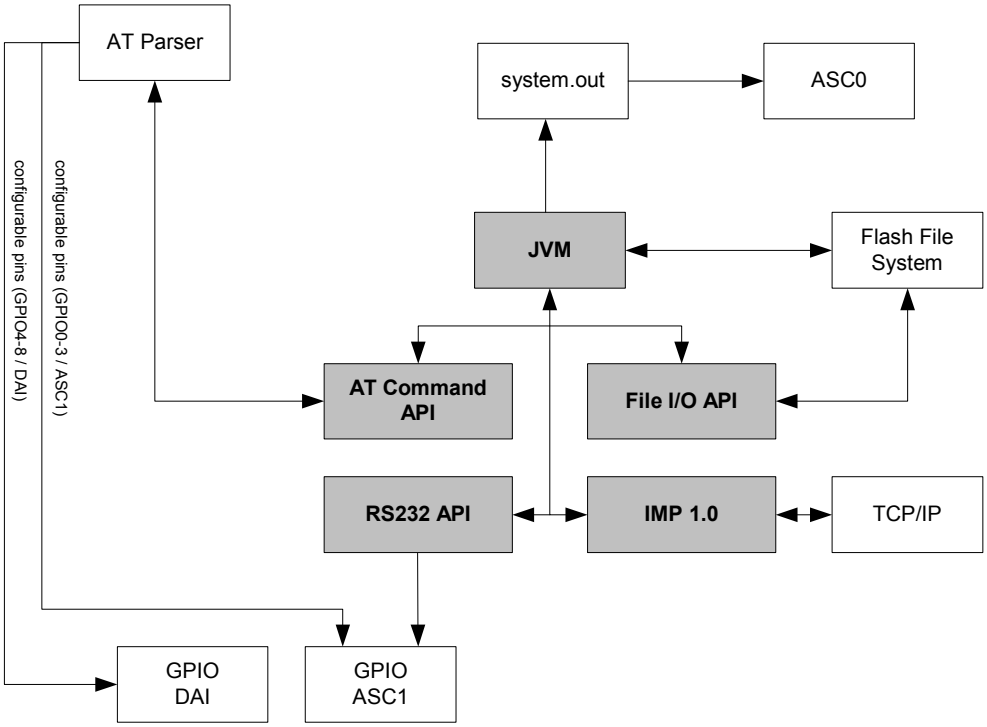


Figure 4: Data flow of a Java application running on the module.

The diagram shows the data flow of a Java application running on the module. The data flow of a Java application running in the debug environment can be found in Figure 15.

The compiled Java applications are stored as JAR files in Flash File System of module. When the application is started, the JVM interprets the JAR file and calls the interfaces to module environment.

The module environment consists of:

- Flash File System: available memory for Java applications
- TCP/IP: module internal TCP/IP stack
- GPIO: general purpose I/O
- DAI: Digital Audio Interface
- ASC1: Asynchronous serial interface 1
- AT parser: accessible AT parser

The Java environment on the module consists of:

- JVM: Java Virtual Machine
- AT command API: Java API to AT parser
- File API: Java API to Flash File System
- RS-232 API: Java API to ASC1
- IMP 1.0: Java API to TCP/IP

4.4 Handling interfaces and data service resources

To develop Java applications the developer must know which resources, data services and hardware access are available.

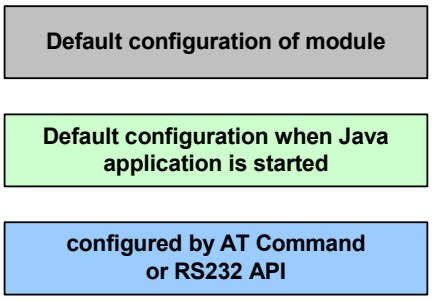
- There are three AT parsers available, one parser with CSD and two without.
- There is hardware access over
  - two serial interfaces: ASC0 (System.out only) and ASC1 (fully accessible).
  - general purpose I/O, (GPIO, divided into two groups of pins, pin group 0 and 1). To configure the hardware access, please refer to the AT Command Set [4] and the Hardware Interface Description [5].
  - All restrictions of combinations are described in section 4.4.1.

- A Java application has:
  - three instances of the AT command class, each of which would, in turn, be attached to one of the three AT parsers.
  - one instance of access to a serial interface, ASC1, through the RS-232 API.
  - System.out over the serial interface, ASC0, for debugging.

4.4.1 Module states

The module can exist in the following six states vis-à-vis a Java application and the four I/O pins, (pin group 0), used for ASC1 and general purpose I/O. See the AT Command Set [4] for information about any AT commands referenced. A state transition diagram is shown in Figure 11.

Pin group 1 and DAI are not mentioned here. This section only shows how Java applications must share AT parsers, GPIO pin group 0 and ASC1 resources.


Default configuration of module
Default configuration when Java application is started
configured by AT Command or RS232 API

Legend of colors in following figures

4.4.1.1 State 1: default – no Java application is running

This is the default state. The Java application is inactive and there is an AT interface with CSD on ASC0. One serial interface is available for AT commands; nothing else is configured.

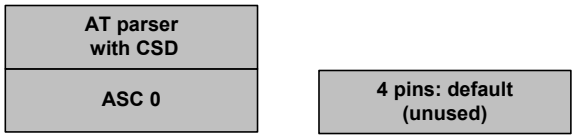


Figure 5: Module State 1

4.4.1.2 State 2: no Java application is running, general purpose I/O

The Java application is inactive. There is an AT parser with CSD on ASC0 and the four I/0 pins are used as general purpose I/O. ASC1 is not configured. The pins are configured by *at^scpin* (refer *to* AT Command Set [4]).

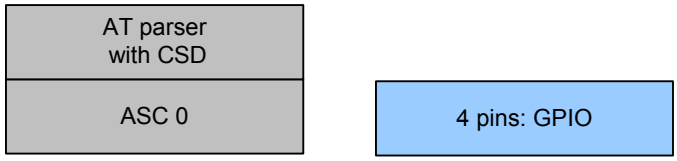


Figure 6: Module State 2

4.4.1.3 State 3: no Java application is running, AT interface on ASC1

The Java application is inactive and there is an AT interface with CSD on ASC0 and a second AT interface (without CSD) on ASC1. The second AT interface is activated by *at^shifm* (refer *to* AT Command Set [4]).

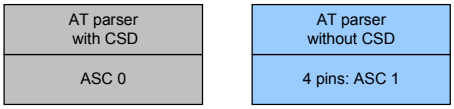


Figure 7: Module State 3

4.4.1.4 State 4: default – Java application active

The Java application is active and ASC0 is used as System.out. Java instances of AT command are connected to the available AT parsers. The Java application is activated with *at^sjra* (refer to AT Command Set [4]) or autostart.

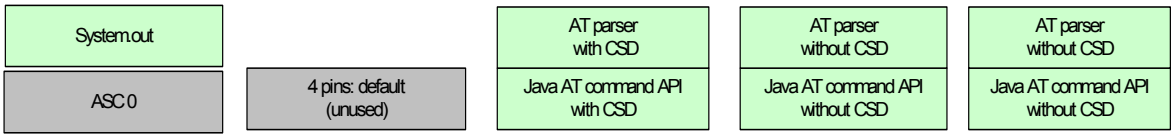


Figure 8: Module State 4

4.4.1.5 State 5: Java application active, general purpose I/O

The Java application is active, ASC0 is used as System.out and the four I/O pins are configured as general purpose I/O. The Java application is activated with *at^sjra.* The I/O pins are configured by *at^scpin*. Refer to the AT Command Set [4] for AT command details.

System.out
ASC0
4 pins: GPIO
AT parser 0 with CSD
Java AT command API with CSD
AT parser without CSD
Java AT command API without CSD
AT parser without CSD
Java AT command API without CSD

Figure 9: Module State 5

4.4.1.6 State 6: Java application active, Java interface to ASC1

The Java application is running, ASC0 is used as System.out and the Java instance of the RS-232 serial interface is connected to ASC1. The Java application is activated with *at^sjra* (refer *to* AT Command Set [4]).

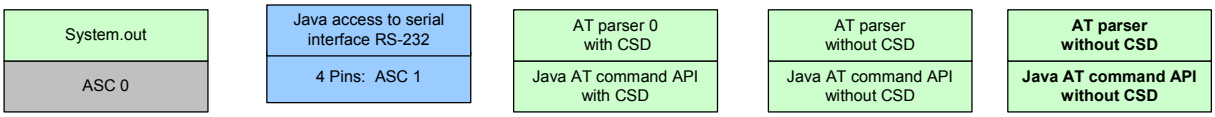

System.out
ASC 0
Java access to serial interface RS-232
4 Pins: ASC 1
AT parser 0 with CSD
Java AT command API with CSD
AT parser without CSD
Java AT command API without CSD
AT parser without CSD
Java AT command API without CSD

Figure 10: Module State 6

4.4.2 Module state transitions

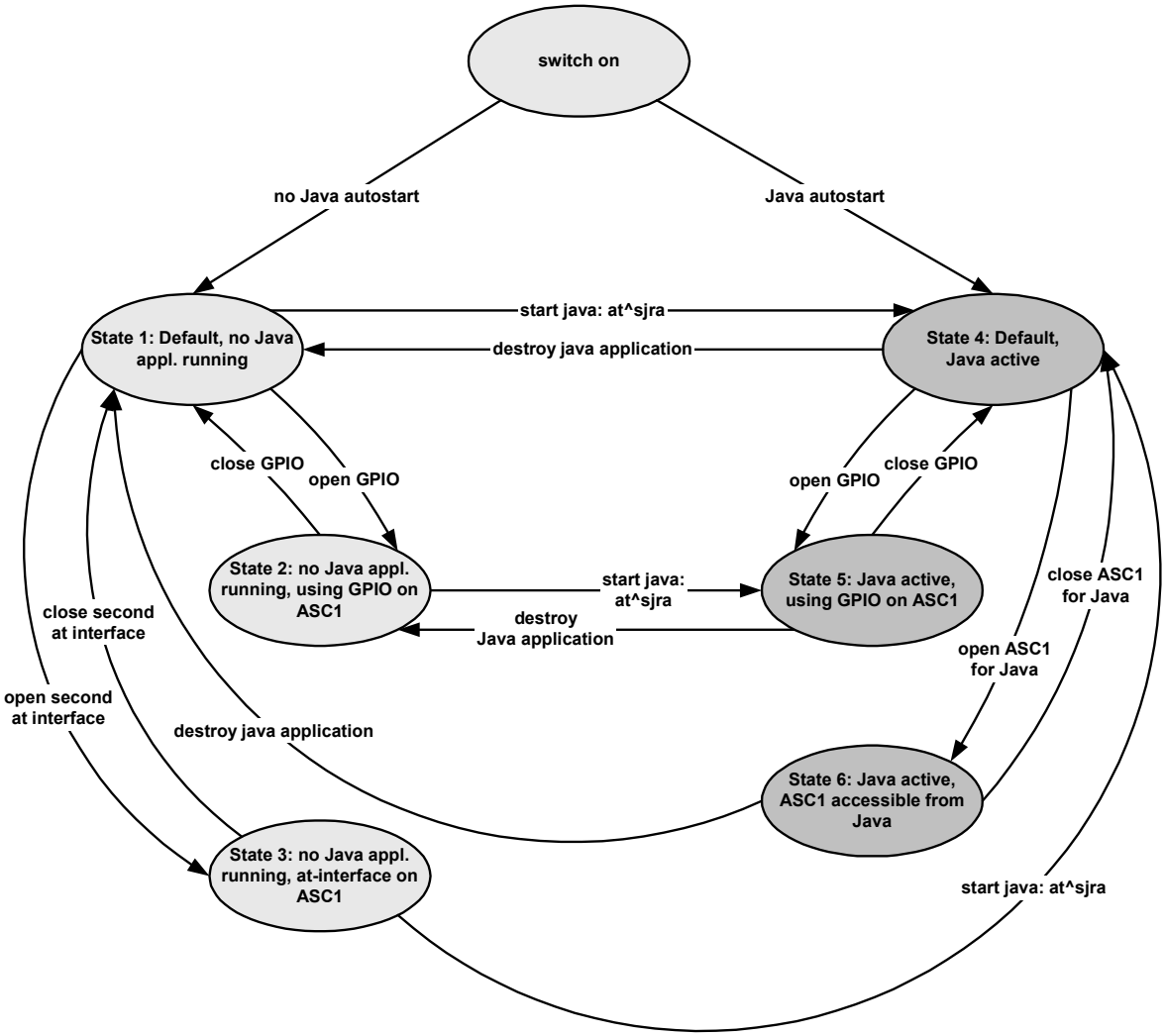

switch on
no Java autostart
Java autostart
State 1: Default, no Java appl. running
start java: at^sjra
destroy java application
State 4: Default, Java active
close GPIO
open GPIO
open GPIO
close GPIO
State 2: no Java appl. running, using GPIO on ASC1
start java: at^sjra
destroy Java application
State 5: Java active, using GPIO on ASC1
close second at interface
close ASC1 for Java
open ASC1 for Java
open second at interface
destroy java application
State 6: Java active, ASC1 accessible from Java
State 3: no Java appl. running, at-interface on ASC1
start java: at^sjra

Figure 11: Module State Transition Diagram

Note: No AT parser is available over serial interface ASC0 or ASC1 while a Java application is running on the module.

- System.out is available on ASC0 for debugging while a Java application is running.
- Comparison of Java AT command API's:

Table 1: Java AT commands

	Voice calls incoming outgoing	Data calls incoming outgoing	SMS incoming outgoing	GPRS connection	Phonebook management	AT commands
Java AT command API with CSD	●	●	●	●[2)]	●	●
Java AT command API without CSD	●	-	●	●[2)]	●	● [1)]

● indicates that the functionality is available from the Java AT command API
--- indicates that the functionality is not available from the Java AT command API
1) except for AT commands related to data calls
2) only two Java AT command API's can be used in parallel to transmit GPRS data

5 MIDlets

The J2ME™ Mobile Information Device Profile (MIDP) provides a targeted Java API for writing wireless applications. The MIDP runs on top of the Connected Limited Device Configuration (CLDC), which in turn, runs on top of the J2ME™. MIDP applications are referred to as MIDlets. MIDlets are controlled by the mobile device implementation that supports the CLDC and MIDP. Since IMP 1.0 is a subset of MIDP 1.0, IMP 1.0 includes MIDlets. The MIDlet code structure is very similar to applet code. There is no main method and MIDlets always extend from the MIDlet class. The MIDlet class in the MIDlet package provides methods to manage a MIDlet's life cycle.

5.1 MIDlet documentation

MIDP and MIDlet documentation can be found at http://wireless.java.sun.com/midp/ and in the html document directory of IMPSIEMENS,
…\Siemens\SMTK\TC45\IMPSiemens\doc\index.html

5.2 MIDlet life cycle

The MIDlet life cycle defines the protocol between a MIDlet and its environment through a simple well-defined state machine, a concise definition of the MIDlet's states and APIs to signal changes between the states. A MIDlet has three valid states:

- **Paused** – The MIDlet is initialised and is quiescent. It should not be holding or using any shared resources. The Paused state in IMP 1.0 is only used during MIDlet construction, it will never reached again after changed state from paused state.
- **Active** – The MIDlet is functioning normally.
- **Destroyed** – The MIDlet has released all of its resources and terminated. This state is only entered once.

State changes are affected by the MIDlet interface, which consists of:
- *pauseApp()* – the MIDlet should release any temporary resources and become passive.
- Note: Because the Paused state is not used in IMP 1.0, this method will never be called.
- *startApp()* – the MIDlet starts it's execution, needed resources can be acquire here or in the MIDlet constructor
- *destroyApp()* – the MIDlet should save any state and release all resources
- Note: destroyApp() is called when a MIDlet should terminate caused by device.
- *notifyDestroyed()* – the MIDlet notifies the application management software that it has cleaned up and is done
- Note: the only way to terminate a MIDlet is to call notifyDestroyed(), but destroyApp() is not automatically called by notifyDestroyed().
- *notifyPaused()* – the MIDlet notifies the application management software that it has paused
  Note: notifyPaused() is not implemented in IMP 1.0.
- *resumeRequest()* – the MIDlet asks application management software to be started again.
  Note: notifyPaused() is not implemented in IMP 1.0.
  - *getAppProperty()* – gets a named property from the MIDlet

Table 2: A typical sequence of MIDlet execution is:

Application Management Software	MIDlet
The application management software creates a new instance of a MIDlet.	The default (no argument) constructor for the MIDlet is called; it is in the Paused state.
The application management software has decided that it is an appropriate time for the MIDlet to run, so it calls the *MIDlet.startApp* method for it to enter the Active state.	The MIDlet acquires any resources it needs and begins to perform its service.
The application management software no longer needs the application be active, so it signals it to stop performing its service by calling the *MIDlet.pauseApp* method. * IMP 1.0 does not need the Paused state except for default constructor for the MIDlet. A *MIDlet.pauseApp* method is not necessary.	The MIDlet stops performing its service and might choose to release some resources it currently holds.
The application management software has determined that the MIDlet is no longer needed, or perhaps needs to make room for a higher priority application in memory, so it signals the MIDlet that it is a candidate to be destroyed by calling the *MIDlet.destroyApp* method.	If it has been designed to do so, the MIDlet saves state or user preferences and performs clean up.

5.3 Hello World MIDlet

Here is a sample HelloWorld program.

```
/**
 * HelloWorld.java
 */

package example.helloworld;
import javax.microedition.midlet.*;
import java.io.*;


public class HelloWorld extends MIDlet {

  /**
   * HelloWorld - default constructor
   */
  public HelloWorld() {
    System.out.println("HelloWorld: Constructor");
  }

  /**
   * startApp()
   */
  public void startApp() throws MIDletStateChangeException {
    System.out.println("HelloWorld: startApp");
    System.out.println("\nHello World!\n");
    notifyDestroyed();
  }

  /**
   * pauseApp()
   */
  public void pauseApp() {
    System.out.println("HelloWorld: pauseApp()");
  }
```

```
  /**
   * destroyApp()
   */
  public void destroyApp(boolean cond) {
    System.out.println("HelloWorld: destroyApp(" + cond + ")");
  }
}
```

6 AT Java Open Framework (AJOF)

6.1 AT Commands

The TC45 AT Command Set [4] contains all of the standard AT commands needed to operate a GSM/GPRS mobile. Please note that no fax commands are supported when the TC45 is operated in the Java environment. Simply issuing individual AT commands can control the TC45 module. See the “AT Command Set” to learn more about AT commands. The AT Java Open Framework allows an application to be built without dealing with individual AT commands. This framework sits on top of the AT command API.

The AT commands have been divided into six functional areas. Each area is represented as a package. Documentation for the methods of each area and the AT commands covered by these methods can be found in *…\Siemens\SMTK\TC45\AJOF\doc\index.html*, [10].

Extensions are possible to this framework. The Java tutorial in Chapter 11 gives examples on how to use this framework.

6.1.1 Mobile engine status

This area abstracts AT commands which affect the ME status, either by setting or returning device parameters. The methods supplied in this category are mostly query methods. Most of the methods in this category can be called regardless of the ME’s current status. The values set by these methods remain valid after the method call is finished.

6.1.2 Voice call handling

This category provides the methods for the handling of voice calls. Some of these methods can only be executed successfully when the module is in a particular state.

6.1.3 CSD call and GPRS connection handling

Similarly to the voice call handling methods, this category provides methods for handling GPRS connections and CSD calls.

6.1.4 Short Message Service

There are two formats, text and PDU, for sending short messages. The developer can create messages of both formats, however, AJOF handles SMs internally in the PDU format. With this framework, sending a short message should be as simple as writing a string to the screen. The methods provided do all the appropriate conversions for reading or preparing an SM. These conversions are transparent to the application programmer.

6.1.5 Phonebook features

The phonebook methods provide access to the phonebook storage media. This class provides methods to select a storage medium, list the stored entries, write into, browse through, read or delete from a storage medium.

6.1.6 Pin I/O

The pin I/O class provides methods for configuring the pins, writing to and reading from the pins, configuring the pins as a port and listening to the pins/port.

7 File transfer to module

7.1 Module Exchange Suite

The Module Exchange Suite allows you to view the Flash file system on the module as a directory from Windows Explorer. Make sure that the module is turned on and that ASC0 of the module is connected to the configured comport of Module Exchange Suite. The adjustment of the configured comport can be checked by attributes on Module directory.
Please note that the Module Exchange Suite can be used only if the module is in normal mode.

7.1.1 Windows based

The directory is called “Module” and can be found at the top level of workspace “MyComputer”. To transfer a file to the module, simply copy the file from the source directory to the target directory in the “Module”.

7.1.2 Command line based

A suite of command line tools is available for accessing the module’s Flash file system. They are installed in the Windows System directory so that the tools are available from any directory. The module’s file system is accessed with *mod:*. The tools included in this suite are MESdel, MEScopy, MESdir, MESmkdir, MESrmdir and MESformat. Entering one of these commands without arguments will describe the command’s usage. The tools mimic the standard directory and file commands. A path inside the module’s file system is identified by using “mod:” instead of a PC drive letter (e.g. “MESdir mod.” lists the contents of the module’s root directory).

To direct the output of the Visual Basic Script (VBS) based command line tools into the command prompt window instead of a Windows popup window the VBS command-scripting must be “used host” instead of the Windows scripting host. This is achieved by calling the scripting host once with the parameter '//H:CScript'. Call the command "cscript //H:CScript" once in a command prompt window. This setting will be stored eternally. To switch back to the wscripting host call "cscript //H:WScript".

7.2 Over the Air Provisioning

See Chapter 8 for OTA provisioning.

7.3 Security issues

The developer should be aware of the following security issues.

7.3.1 Module Exchange Suite

The serial interface should be mechanically protected.

7.3.2 OTAP

- A password should be used to update with OTA (SMS Authentication)
- Parameters should be set to fixed values (*at^sjotap*) whenever possible so that they cannot be changed over the air.
- The http server should be secure. (e.g. Access control via basic authentication)

8 Over The Air Provisioning (OTAP)

8.1 Introduction to OTAP

OTA (Over The Air) Provisioning of Java Applications is a common practice in the Java world. OTAP describes mechanism to install, update and delete Java applications over the air. Normally this means that a user controls this mechanisms by interaction (keys, display) with the Java device, e.g. as described in the MIDP specification (http://java.sun.com/products/midp/OTAProvisioning-1.0.pdf). The TC45 product currently implements the IMP1.0 which is based on MIDP1.0, so the OTAP procedure is currently also based on the MIDP1.0 recommended practice for Over The Air Application Provisioning.

The OTAP mechanism described in this document does not require any physical user interaction with the device; it can be fully controlled over the air interface. Therefore it is suitable for Java devices that are not supposed to have any manual interaction like vending machines or electricity meters.

8.2 OTAP overview

To use OTAP, the developer needs, apart from the device fitted with the TC45 module, an http server, which is reachable through a TCP/IP connection either over GPRS or CSD, and a SMS sender, which can send Class1, PID $7d short messages. This is the PID reserved for module's data download.

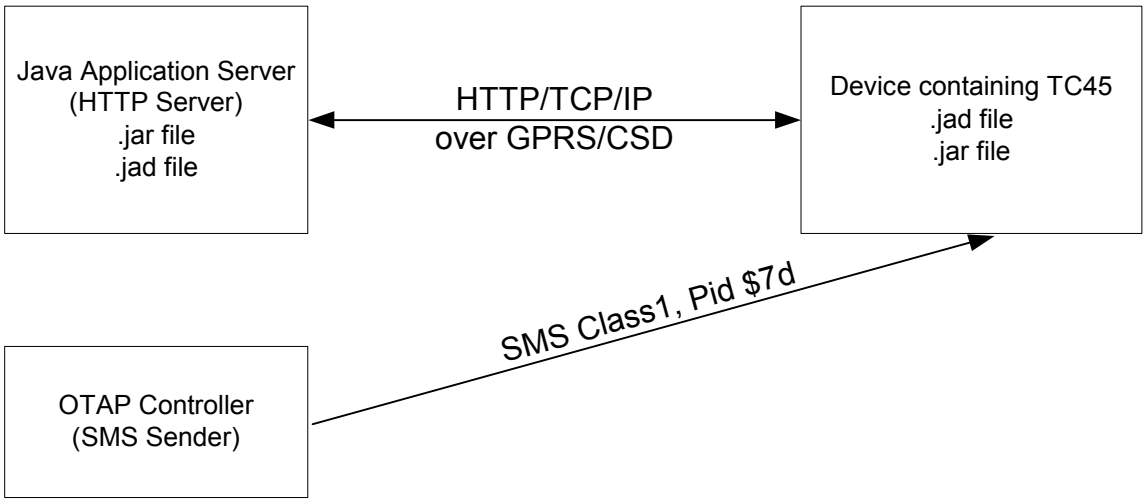


Figure 12: OTAP Overview

The Java Application Server (http Server) contains the .jar and the .jad file, which are to be loaded on the device. Access to these files can be protected by http basic authentication.
The OTAP Controller (SMS Sender) controls the OTAP operations. It sends SMs, with or without additional parameters, to the devices that are to be operated. These devices then try to contact the http server and download new application data from it. The OTAP Controller will not get any response about the result of the operation. Optionally the server might get a result response through http.

There are two types of OTAP operations:

- Install/Update: A new JAR and JAD file are downloaded and installed.
- Delete: A complete application (.jar, .jad, all application data and its directory) is deleted.

8.3 OTAP Parameters

There is a set of parameters that control the OTAP procedures. These parameters can either be set by AT command (*at^sjotap*, refer to AT Command Set [7]) presumably during the production of the device, or by SM (see Section 8.4) during operation of the device in the field. None of the parameters, which are set by AT command, can be overwritten by SM.

- JAD File URL: the location of the JAD file is used for install or update procedures. The JAD file needs to be located on the net (e.g. http://someserver.net/somefile.jad or http://192.168.1.2/somefile.jad).
- Application Directory: this is the directory where a new application (JAD and JAR file) is installed. The delete operation deletes this directory completely.
- http User: a username used for authentication with the http server.
- http Password: a password used for authentication with the http server.
- Bearer: the network bearer used to open the HTTP/TCP/IP connection, either GPRS or CSD.
- APN or Number: depending on the selected network bearer this is either an access point name for GPRS or a telephone number for CSD.
- Net User: a username used for authentication with the network.
- Net Password: a password used for authentication with the network.
- DNS: a Domain Name Server's IP address used to query hostnames.

There is one additional parameter that can only be set by AT command:

- SM Password: it is used to authenticate incoming OTAP SMs. Setting this password gives an extra level of security.
  Note: If there was a password set by AT command, all SMs have to include this password

Table 3: Parameters and keywords

Parameters	Max. Length AT	Keyword SM	Install/update	delete
JAD File URL	100	JADURL	mandatory	unused
Application Directory	50	APPDIR	mandatory	mandatory
HTTP User	32	HTTPUSER	optional	unused
HTTP Password	32	HTTPPWD	optional	unused
Bearer	--	BEARER	mandatory	unused
APN or Number	65	APNORNUM	mandatory for CSD	unused
Net User	32	NETUSER	optional	unused
Net Password	32	NETPWD	optional	unused
DNS	--	DNS	optional	unused
SM Password	32	PWD	optional	optional

The length of the string parameters in the AT command is limited (see Table 3), the length in the SM is only limited by the maximum SM length.
The minimum set of required parameters depends on the intended operation (see Table 3).

8.4 Short message format

An OTAP control SM must be a Submit PDU with Class1, PID $7d and 8 bit encoding. As a fallback for unusual network infrastructures the SM can also be of Class0 and/or PID $00. The content of the SM consists of a set of keywords and parameter values. These parameters might be distributed over several SMs. There is one single keyword to start the OTAP procedure. For parameters that are repeated in several SMs only the last value sent is valid. For example, an SM could look like this:

Install operation:

First SM:
```
OTAP_IMP1.0
PWD:secret
JADURL:http://www.greatcompany.com/coolapps/mega.jad
APPDIR:a:/work/appdir
HTTPUSER:user
HTTPPWD:anothersecret
```

Second SM:
```
OTAP_IMP1.0
PWD:secret
BEARER:gprs
APNORNUM:access.to-thenet.net
NETUSER:nobody
NETPWD:nothing
DNS:192.168.1.2
START:install
```

Delete operation:

```
OTAP_IMP1.0
PWD:secret
APPDIR:a:/work/appdir
START:delete
```

The first line is required: it is used to identify an OTAP SM. All other lines are optional and their order is insignificant, each line is terminated with an LF: '\n' even the last one. The keywords, in capital letters, are case sensitive. A colon separates the keywords from their values.

The values of APPDIR, BEARER and START are used internally and have to be lower case. The password (PWD) is case sensitive. The case sensitivity of the other parameter values depends on the server application or the network. It is likely that not all parameters can be sent in one SM. They can be distributed over several SMs. Of course, every SM needs to contain the identifying first line and the PWD parameter, if necessary. The OTAP is started when the keyword START, possibly with a parameter, is contained in the SM and the parameter set is valid for the requested operation. It always ends with a reboot, either when the operation is completed, an error occurred, or the safety timer expired. This also means all parameters previously set by SM are gone.

Apart from the first and the last line in this example, these are the parameters described in the previous section. Possible parameters for the START keyword are: “install”, “delete” or nothing. In the last case, an install operation is done by default.

The network does not guarantee the order of SMs. So when using multiple SMs to start a OTAP operation their order on the receiving side might be different from the order in which they were sent. This could lead to trouble because the OTAP operation probably starts before all parameters are received. If you discover such problems, try to wait a few seconds between sending the SMs.

8.5 Java file format

In general, all Java files have to comply with the IMP 1.0 and TC45 specifications. There are certain components of the JAD file that the developer must pay attention to when using OTAP:

- MIDlet-Jar-URL: make sure that this parameter points to a location on the network where your latest JAR files will be located, e.g. http://192.168.1.3/datafiles/mytest.jar, not in the filesystem like file://a:/java/mytest/mytest.jar. Otherwise this JAD file is useless for OTAP.
- MIDlet-Install-Notify: this is an optional entry specifying a URL to which the result of an update/install operation is posted. That is the only way to get any feedback about the outcome of an install/update operation. The format of the posted URL complies with the MIDP OTA Provisioning specification. In contrast to the jar and jad file this URL must not be protected by basic authentication.
- MIDlet-Name, MIDlet-Version, MIDlet-Vendor: are mandatory entries in the JAD and Manifest file. Both files must contain equal values, otherwise result 905 (see 8.7) is returned.
- MIDlet-Jar-Size must contain the correct size of the jar file, otherwise result 904 (see 8.7) is returned.

Example:
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: TLR Inc.
MIDlet-Jar-URL: http://192.168.1.3/datafiles/MyTest.jar
MIDlet-Description: My very important test
MIDlet-1: MyTest, , example.mytest.MyTest
MIDlet-Jar-Size: 1442

A suitable Manifest file for the JAD file above might look like:

```
Manifest-Version: 1.0
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: TLR Inc.
MIDlet-1: MyTest, , example.mytest.MyTest
```

8.6 Procedures

8.6.1 Install/Update

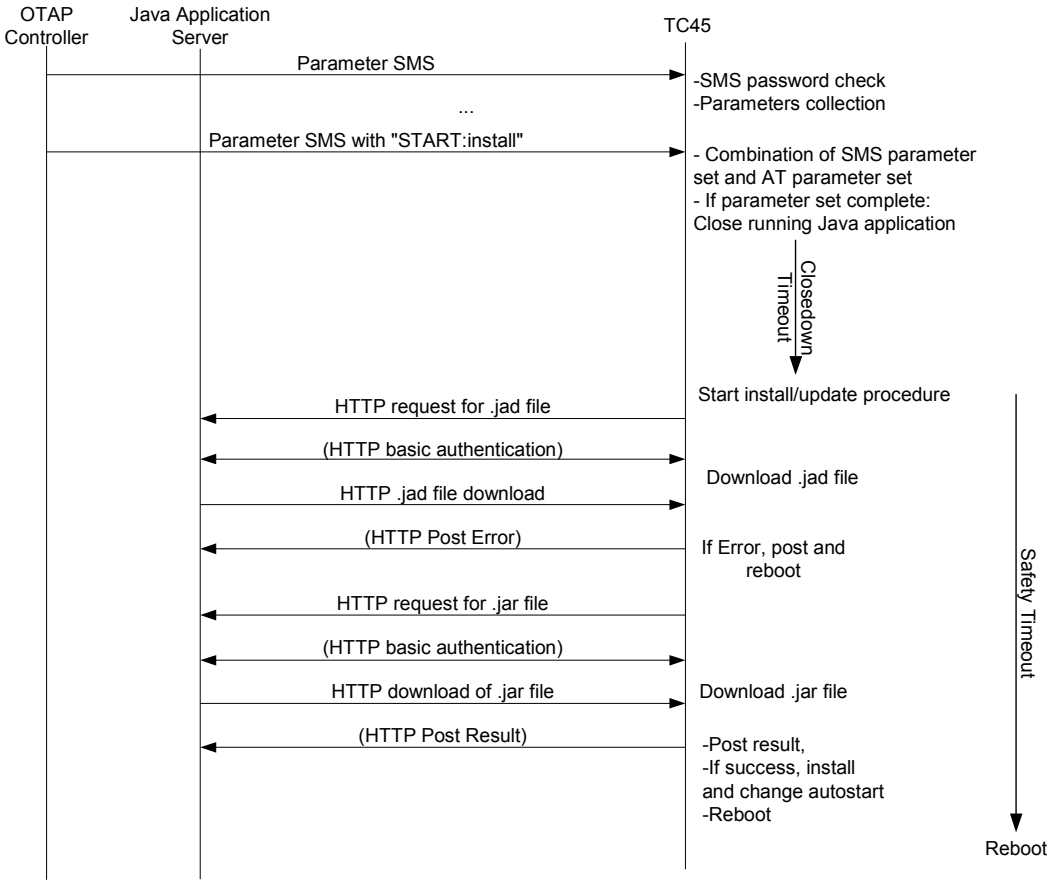


Figure 13: OTAP: Install/Update Information Flow
The messages in brackets are optional

When an SM with keyword START:install is received and there is a valid parameter set for the operation, the module always reboots either when the operation completed, an error occurred or the safety timer expired. If there is any error during an update operation the old application is kept untouched, with one exception. If there is not enough space in the file system to keep the old and the new application at the same time, the old application is deleted before the download of the new one, therefore it is lost when an error occurs.
If install/update was successful the autostart is set to the new application.

8.6.2 Delete

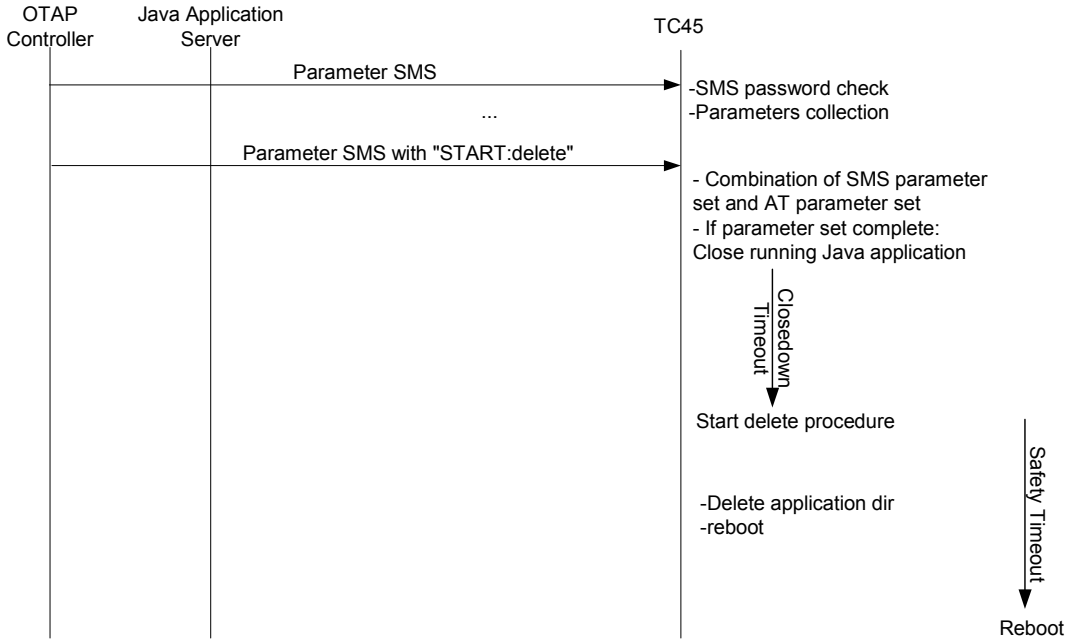


Figure 14: OTAP: Delete Information Flow

When an SM with keyword START:delete is received and there is a valid parameter set for this operation, the module reboots either when the operation completed, an error occurred or the safety timer expired. If there is any error the application is kept untouched. Autostart is not changed. No result code is passed back.

8.7 Time out values and result codes

Timeouts:
- Closedown Timeout: 10 seconds
- Safety Timeout: 5 minutes

Result Codes: Supported status codes in body of the http POST request:
- 900 Success
- 901 Insufficient memory in filesystem
- 902 -not supported-
- 903 -not supported-
- 904 JAR size mismatch, given size in JAD file does not match real size of jar file
- 905 Attribute mismatch, one of the mandatory attributes MIDlet-name, MIDlet-version, MIDlet-Vendor in the JAD file does not match those given in the JAR manifest
- 906 -not supported-

8.8 Notes

- For security reasons it is recommended to use a SMS password. Otherwise the “delete” operation can remove whole directories without any authentication.
- For extra security, setup a private CSD/PPP Server and set its phone number as a fixed parameter. This way, applications can only be downloaded from one special server.
- As a side effect, OTAP can be used to simply reboot the module. Just start an OTAP procedure with a parameter set which does not really do anything, like a delete operation on a nonexistent directory.
- If you don’t want to start OTAP by SMS let your Java application do it by issuing the *at^sjotap* command. That triggers a install/update operation as described in chapter 8.6.1 but without the SMS part.
  Note: If a malfunctioning Java application is loaded the SM method will still be needed for another update.
- The OTAP procedure cannot be tested in the debug environment
- Be aware that the module needs to be booked into the network to do OTAP. That means that either the Java application has to enter the PIN or the PIN needs to be disabled. To ensure that you can still do OTAP even, or especially, when the current Java application does not work properly, disable the PIN during OTAP activities. This might not be possible in all networks.
- The OTAP procedure might fail due to call collision, e.g. a incoming call when OTAP tries to start a CSD connection.

8.9 How to

This chapter is supposed to be a step-by-step guideline for using OTAP.

1. Do you need OTAP? Is there any chance that it might be necessary to update the Java application, install a new one or delete it? The reason might be that your device is in the field and you cannot or do not want to do it over the serial line. If the answer is “yes” then read through the following steps, if the answer is “no” then just consider setting the OTAP SMS password to protect your system. Then you are done with OTAP.
2. Take a look at the parameters (chapter 8.3), which control OTAP. You have to decide which of them you want to allow to be changed over the air (by SMS) and which you do not. This is mainly a question of security and what you can fit into a short message. Then set the “unchangeable” parameters with the AT command (*at^sjotap*).
3. Prepare the http server. The server must be reachable by your device over TCP/IP. That means there is a route from your device over the air interface to the http server and back. When in doubt, write a small Java application using the httpConnection Interface to test it.
4. Prepare the JAR and JAD files which are to be loaded over the air. Make sure that these files conform to the requirements named in chapter 8.5 and that they represent a valid application which can be started by *at^sjra*.
5. Put the files (JAR and JAD) on the http Server. The files can either be publicly available or protected through basic authentication. When in doubt try to download the files from the server by using a common web browser on a PC, which can reach your http server through TCP/IP.
6. Prepare the SMS sender. The sender must be able to send SMs, which conform to chapter 0, to your device. When in doubt try to send “normal” SMs to your device which can than be read out through the at command interface.
7. Test with a local device. Send a suitable short message to your device, which completes the necessary parameter, set and starts the operation. The operation is finished when the device reboots. You can now check the content of the file system, if the correct jar and jad file was loaded into the correct location.
8. Analyze error. If the above test failed, looking at your devices behavior and your http servers access log can give you some hints on what went wrong:
   - If the device did not terminate the running Java application and did not reboot, not even after the safety timeout, either your SM was not understood (probably wrong format) or did not properly authenticate (probably wrong password) or your parameter set is incomplete for the requested operation.
   - If the device terminated the running Java application, but did not access your http server, and rebooted after the safety timeout, there were most likely some problems when opening the network connection. Check your network parameters.
   - If the device downloaded the jad and probably even the jar file but then rebooted without saving them in the file system, most likely one of the errors named in chapter 8.5 occurred. These are also the only error conditions, which can also be reported back. They are posted to the http server if the jad file contains the required URL.
9. Start update of remote devices. If you were able to successfully update your local device, which is hopefully a mirror of all your remote devices, you can start the update of all other devices.

9 Compiling and running a program without a Java IDE

This chapter explains how to compile and run a Java application without a Java IDE. Compiling and running Java applications with a Java IDE is explained in Section 10.2.1.5 and Section 10.2.2.5. Three different methods for running an application are described. The first method explains how to run the program on a PC, that is, in the development environment. The second and third methods describe how to run the application on the actual module.

9.1 Build results

A JAR file has to be created by compiling an SMTK project. A JAR file will contain the class files and auxiliary resources associated with an application. A JAD file contains information (file name, size, version, etc.) on the actual content of the associated JAR file. It must be written by the user. The JAR file has the ".jar" extension and the JAD file has the ".jad" extension. When the module is provisioned with the Module Exchange Suite, as described in Section 7.1, only the JAR file is needed. The JAD file is additionally necessary for OTA provisioning. OTA provisioning is described in Chapter 8.

Note: The path to your installed SIEMENS\SMTK\TC45\IMPSiemens\lib\classes.zip is needed to build a JAR file.

9.1.1 JAD file

In addition to class and resource files, a JAR file contains a manifest file, which describes the contents of the JAR. The manifest has the name manifest.mf and is automatically stored in the JAR file itself. An IMP manifest file for:

```
package example.mytest;
public class MyTest extends MIDlet
```

includes at least:

```
Manifest-Version: 1.0
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: Siemens
MIDlet-1: MyTest, example.mytest.MyTest
```

A JAD file must be written by the developer and must include at least:

```
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: Siemens
MIDlet-1: MyTest, example.mytest.MyTest
MIDlet-Jar-URL: http://192.168.1.3/datafiles/MyTest.jar
MIDlet-Jar-Size: 1408
```

A detailed description of these attributes and others can be found in the Java/MIDlet documentation http://java.sun.com/j2me/docs/alt-html/WTK104_UG/Ap_Attributes.html

9.2 Compile and run on a PC

- Launch a *Command Prompt*. This can be done from the *Programs* menu or by typing "cmd" at the *Run…* prompt in the *Start* menu.
- Change to the directory where the code to be compiled is kept.
- Compile the program with the SDK. An example of a build batch file can be in a sample program directory found in the examples directory, *…\Siemens\SMTK\TC45\IMPSiemens\src\examples.*
- If the compile was successful the program can be run from the command line. Example of a run batch file can be found in the examples directory listed above.

The batch files for compiling and running the samples are using the system environment variables IMP_JDK_DIR that points to the root directory of the installed JDK and IMP_DIR which points to the root directory of the Siemens-SMTK-TC45-IMP installation. The installation process sets these environment variables. A modification is usually not necessary. They might be modified (e.g. for switching to a different JDK) via the advanced system properties as requested.

In this scenario, any AT commands issued by the application are sent to the module by the emulator over ASC0.

9.3 Run on the module with manual start

- The application can be compiled at the prompt as discussed in Section 9.2 or in an IDE.
- Transfer the .jar and/or .jad file from the development platform to the desired directory on the module using the Module Exchange Suite or OTA provisioning. Chapter 7 explains how to download your application to the module.
- Start a terminal program and connect to ASC0.
- The command *at^sjra* is used to start the application and is sent to the module via your terminal program. Either the application can be started by .jar or by .jad file. When you start by .jad file the values of parameters have to be correct.

Example:
In your terminal program enter: *at^sjra=a:/java/jam/example/helloworld/helloworld.jar*
If you prefer to start with jad-File: *at^sjra=a:/java/jam/example/helloworld/helloworld.jad*

The Flash file system on the module is referenced by "a:".

9.4 Run on the module with Autostart

- The application can be compiled at the prompt as discussed in Section 9.2 or in an SMTK integrated IDE.
- Transfer the .jar and/or .jad file from the development platform to the desired directory on the module using the Module Exchange Suite or OTA provisioning. See Chapter 7.

9.4.1 Switch on Autostart

- There is an AT command, *at^scfg*, to configure the autostart functionality. Please refer to the AT Command Set [4].
- Restart the module.

9.4.2 Switch off Autostart

To switch off autostart functionality there are two possibilities:
- at command *at^scfg*
- tool “autostart_off.exe” (included in the Installation CD software)

To disable the automatic start of a user application in a module these steps have to be done:
1. Connect the module to the PC
2. Make sure, that the module is switched off
3. Start the Autostart_Off program
4. Select the COM-Port
5. Press the “Autostart Off” button

10 Debug environment

Please note that this section is not intended as a tutorial in debugging or how to use Sun ONE Studio or Borland JBuilder. Documents for these IDEs can be found on their respective homepages. Once the proper emulator has been selected (as described in the relevant IDE sections below), your Java application can be built, debugged and executed as usual.

10.1 Data flow of a Java application in the debug environment

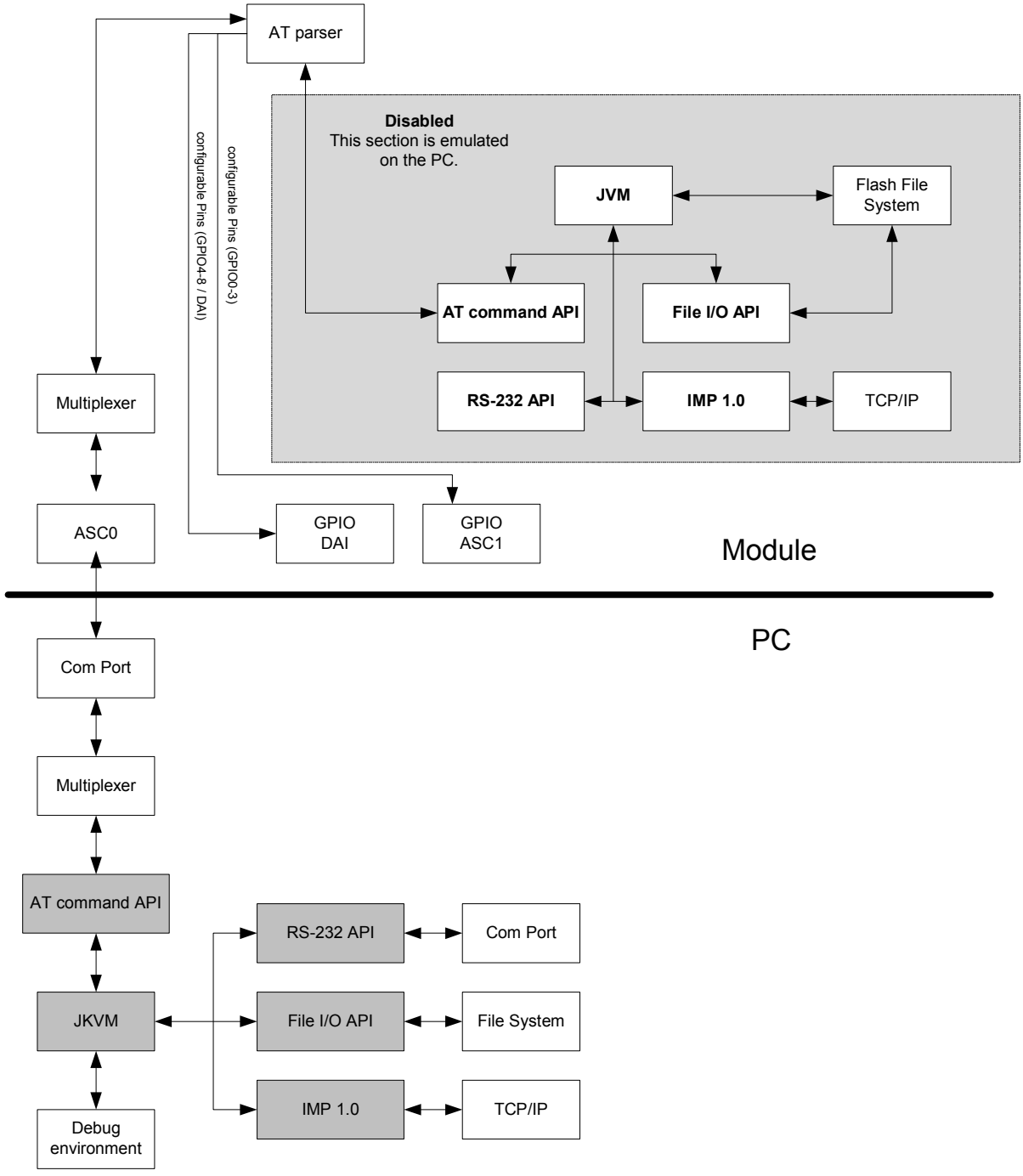


Figure 15: Data flow of a Java application in the debug environment

In the debug environment the module is emulated on the PC and only the AT commands are run on the module, via serial interface. If the I/O pins are configured as ASC1 they are emulated on the PC, shown connected to the RS-232 API in the diagram above, Figure 15. Further information about this configuration is given in Section 10.1.3. Regular pin I/O, however, is controlled on the module via AT commands.

10.1.1 Serial port and pin I/O handling differences

Please note that there are some differences in serial port handling between MIDlets running inside the module and running on the PC debug environment. Obviously there might be some differences in timing issues and buffer sizes. Additionally, pin I/O is done on the same serial port inside the module, regardless of running the application on PC or on module. Therefore, pin I/O and serial port accesses have some restrictions on resources like those described in section 4.4. Creating a CommConnection will fail in an application running on module if pin I/O is already opened and vice versa. Because pin I/O works via AT commands and serial port access via an additional COM port both are not mutually locked in the PC debug environment and both are possible at the same time. Please keep in mind that this does not mean that it is also possible inside the module.

10.1.2 Restrictions

Please note that the PC debug environment only emulates the virtual machine inside the module. Much effort has been taken to ensure that both virtual machines behave in the same way but accordingly, since they are completely different systems, there might be some differences nevertheless. Besides the obvious differences in timing and execution speed there are also differences in file system handling. The Flash file system of the module is simulated in a special sub directory of the PC debug environment, so there are differences in the possible length of file names and also in management of the available size. When using http connections it is the GPRS stack inside the module that is used. This is not available on the PC, so in the debug environment it is the internal IP stack of the operating system that is used. Therefore, the PC must have a working Internet connection for using the http classes.

10.1.3 RS-232 emulator

As stated previously, ASC1 is emulated on the PC in the debug environment. It uses the next available com port after the one used for ASC0. If you wish another com port to use, the WMDebug.ini file must be manually updated. WMDebug.ini can be found in \Siemens\SMTK*TC45*\IMPSiemens\bin. The line *Port1* under *[Ports]* must be altered.

10.2 Java IDE

The SMTK is integrated into your Java IDE during installation. Please note that the IDE integration is intended for debugging purposes using the PC emulator. JAR files used in the module must be configured according to the batch file examples given. If the SMTK install succeeded, one should be able to easily switch between the Siemens environment and Standard-JDK environment. Means that the special libraries/APIs are available, the emulators are available, AT commands sent to module. Regular function of the IDE for non-Siemens projects is unchanged.

10.2.1 Sun ONE Studio 4.1 Mobile Edition

This section indicates the changes to your IDE that you will see after the integration of the SMTK and how to exploit these features to build and debug your applications.

In the *Runtime Explorer*, Figure 16, the installed emulators can be seen under the *Device Emulator Registry*. The *Default Emulator* is not affected by the integration.

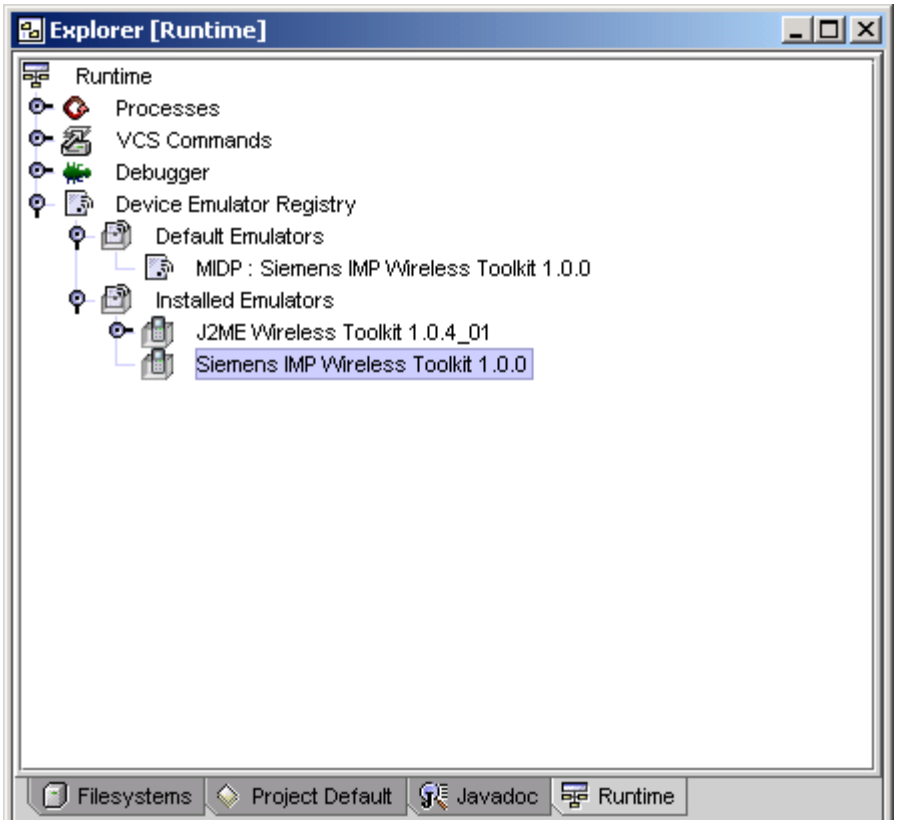


Figure 16: Sun ONE Studio - The installed emulators.

10.2.1.1 Switching emulators

You can easily switch to the SMTK emulator by opening the *Tools* menu, selecting *Siemens TC45* and choosing *Switch Emulator-->Siemens IMP Wireless Toolkit 1.0*, see Figure 3. The default emulator can be switched back to by choosing *Switch Emulator->J2ME Wireless Toolkit*. Any projects built and run when the Siemens emulator is selected will be compiled and run with the Siemens emulator.

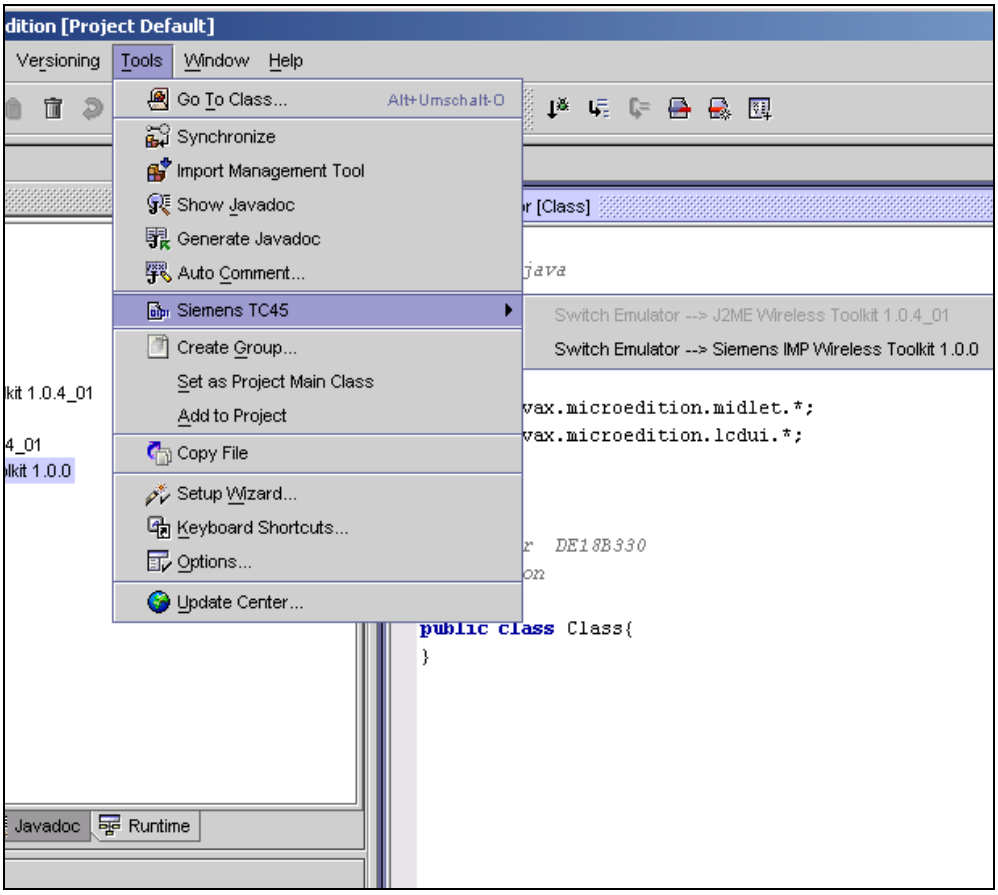


Figure 17: Sun ONE Studio - Switching to the Siemens emulator.

10.2.1.2 Projects

After integration, there are two new projects in the *Project Manager*. The first is a sample HelloWorld project and the second is an empty project that the developer can begin using or use as a model for a new TC45 project. The *Project Manager* is accessed through the *Project* menu.

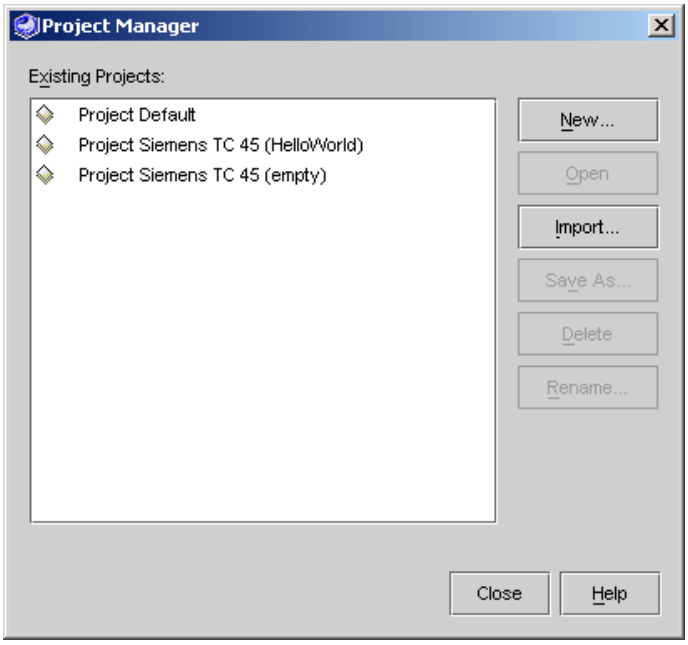


Figure 18: Sun ONE Studio - Project Manager

10.2.1.3 Templates

Templates for a Siemens MIDlet and Java Class can be found in the file explorer and under File->New. The MIDlet template provides the skeleton of a MIDlet application. The Class template can be used to write a class for a Siemens MIDlet application for the TC45.

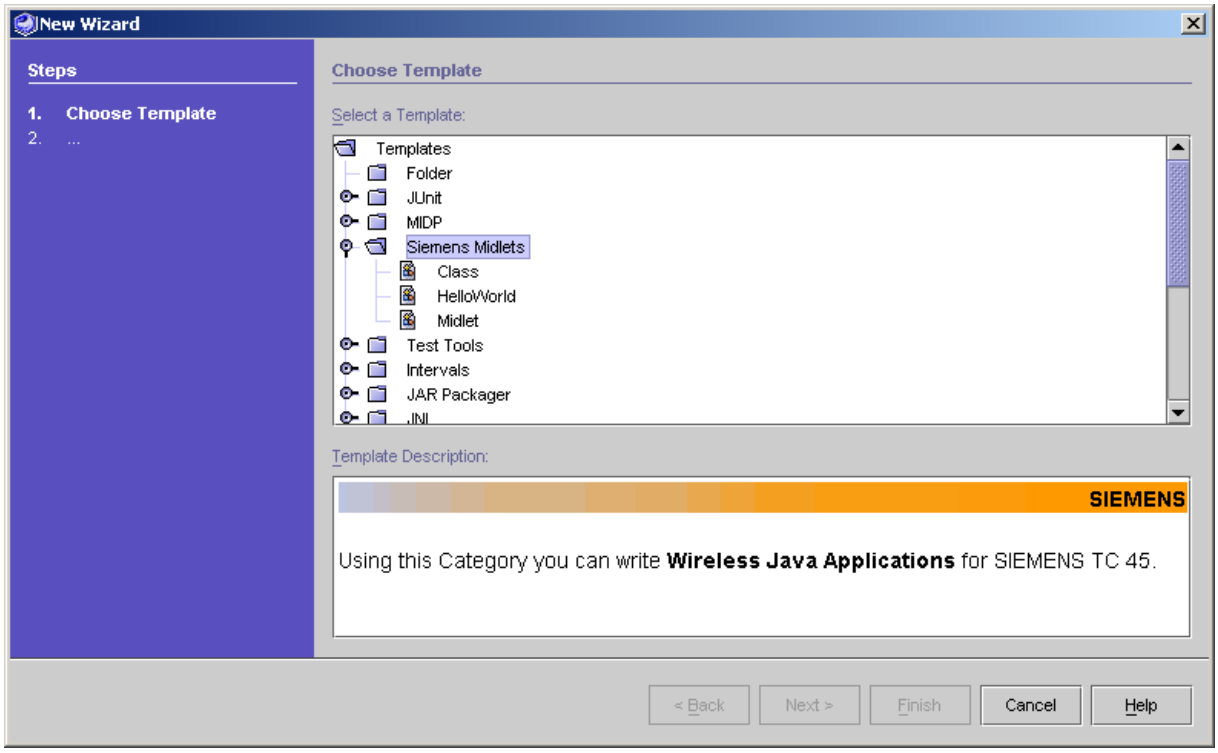


Figure 19: Sun ONE Studio - Selecting a template

10.2.1.4 Examples

There are sample MIDlets in the …/IMPSiemens/src/examples directory. This directory is mounted in Sun ONE Studio and can be seen in the *Explorer* window.

10.2.1.5 Compile and run

Ensure that the proper emulator will be used and compile the project as normal. Any output will be shown in the output window in the IDE.

10.2.2 JBuilder Mobile Edition

10.2.2.1 TC45 SMTK in Personal Edition

JBuilder Personal Edition can handle only one fixed emulator, this is the reason for an integration that is more elaborate than for Enterprise Edition.

10.2.2.1.1 Integration of TC45 SMTK

After integration of TC45 SMTK into JBuilder Personal Edition user must configure the JDK being used according to following steps:

1. Start Jbuilder Personal Edition
2. Open dialog box *Tools→Configure JDKs*: (see Figure)
3. Change JDK by selecting *User Home*. Press *Change* and set path to *<installation directory TC45>\IMPSiemens* and complete configuration by pressing *OK*.

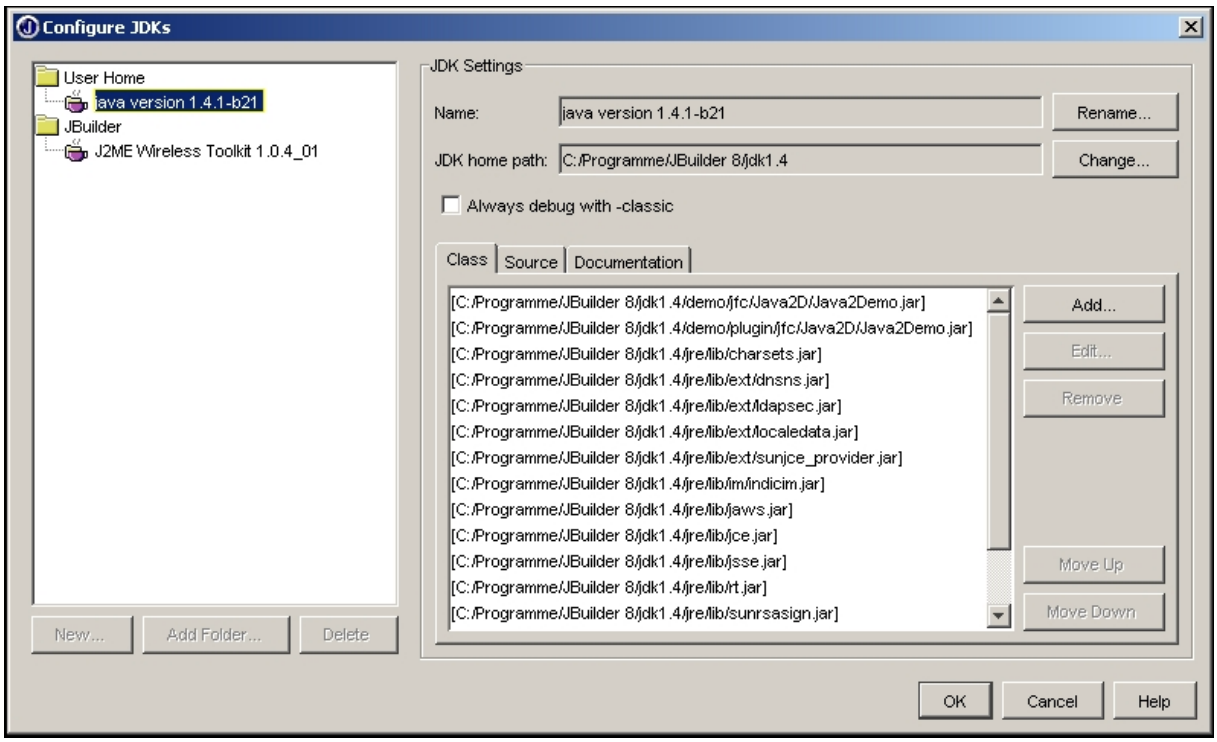


Figure 20: Different JDK must be set in *User Home*

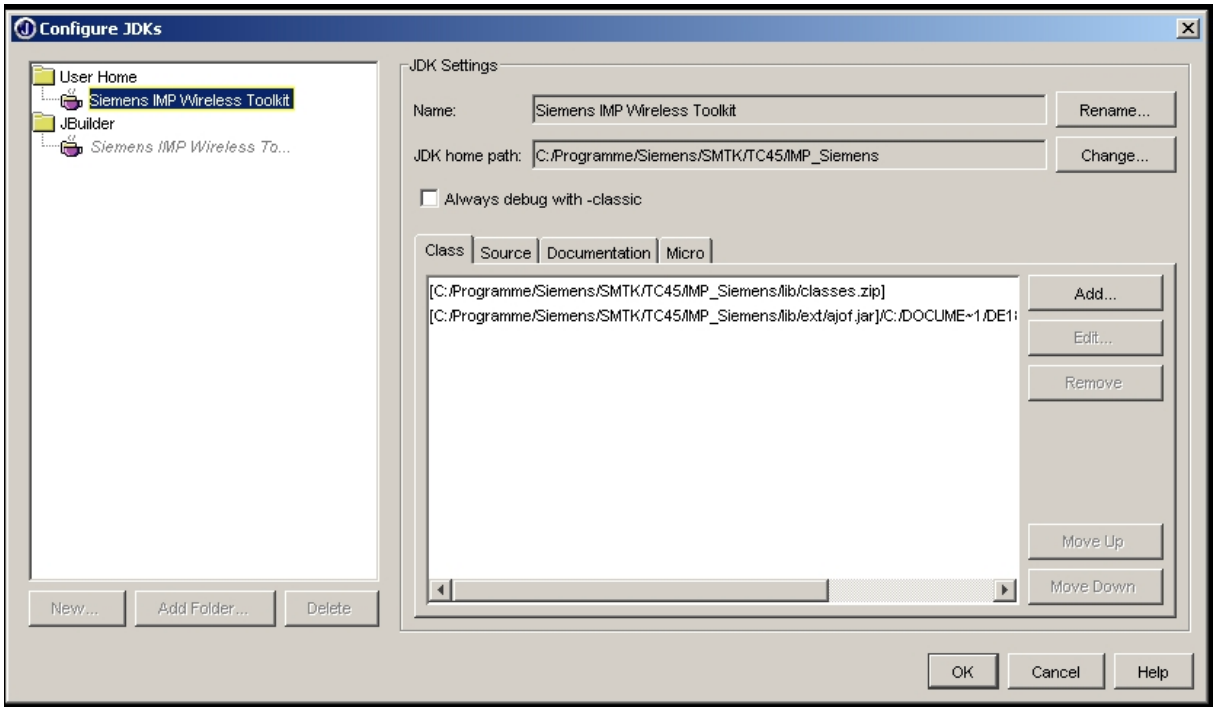


Figure 21: Dialog box after changing the JDK to *Siemens IMP Wireless Toolkit.*

After closing dialog box the *Siemens IMP Wireless Toolkit* will be used by JBuilder Personal Edition as system wide configuration.

10.2.2.1.2 Restoring original settings

For user's convenience the original settings of JBuilder Personal Edition at initial start up is shown below.

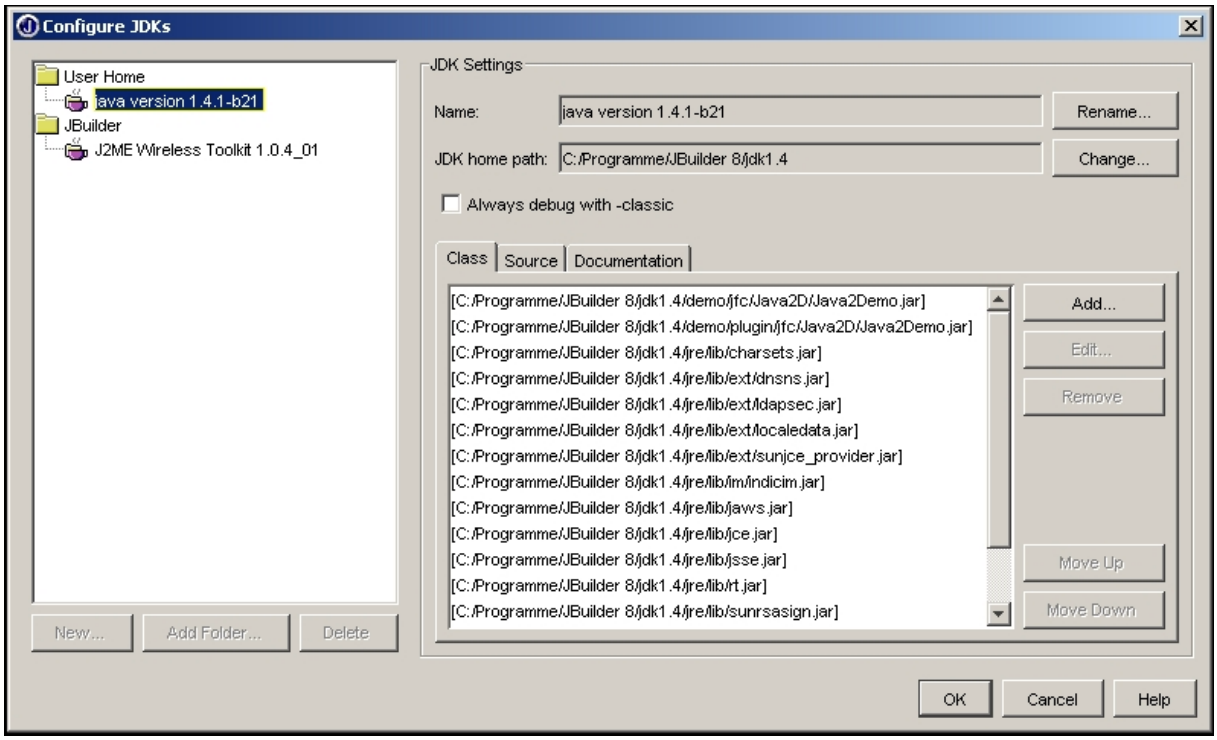


Figure 22: Original setting for *User Home*

To switch back to JDK shipped with JBuilder press *Change* and select location as shown in figure above.

Note: *JDK home path* shown in this figure depends on user's installation directory.

10.2.2.2 TC45 SMTK in Enterprise Edition

The Siemens emulator is installed as a second emulator in JBuilder. Similarly to the Sun ONE Studio integration, the default emulator of JBuilder is untouched. The Siemens emulator settings can be examined by opening Tools->Configure JDKs…

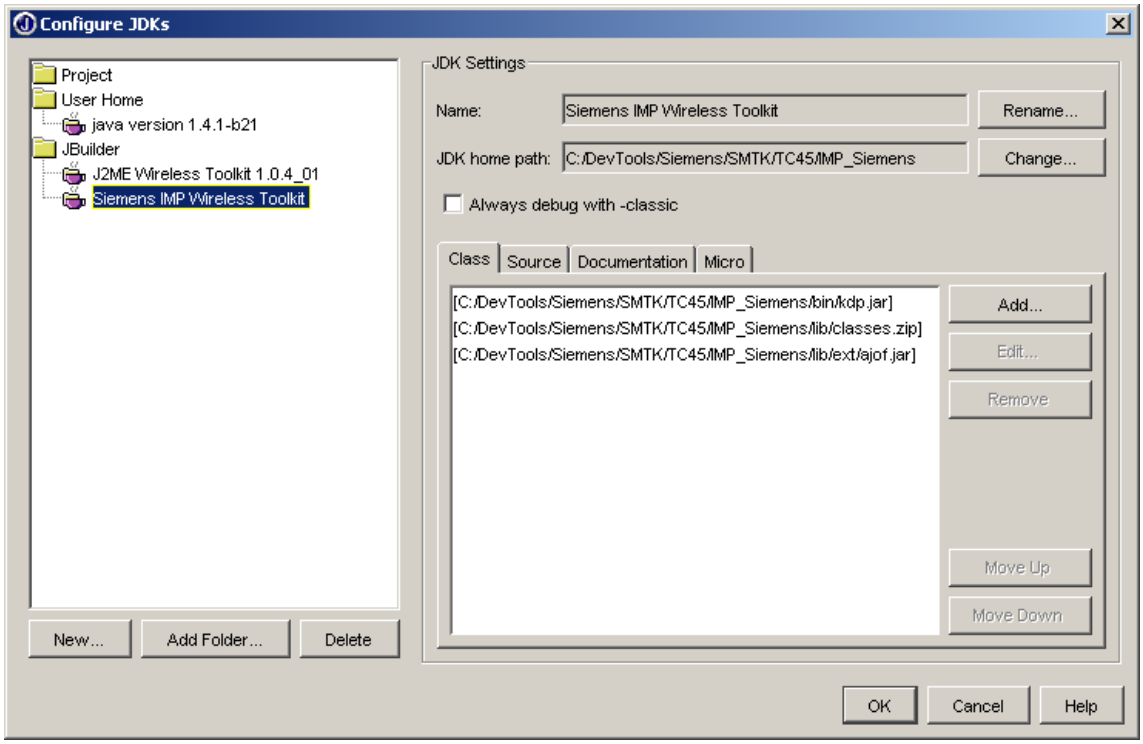


Figure 23: JBuilder - JDK settings

The libraries included with the Siemens emulator can be examined by opening Tools->Configure Libraries…

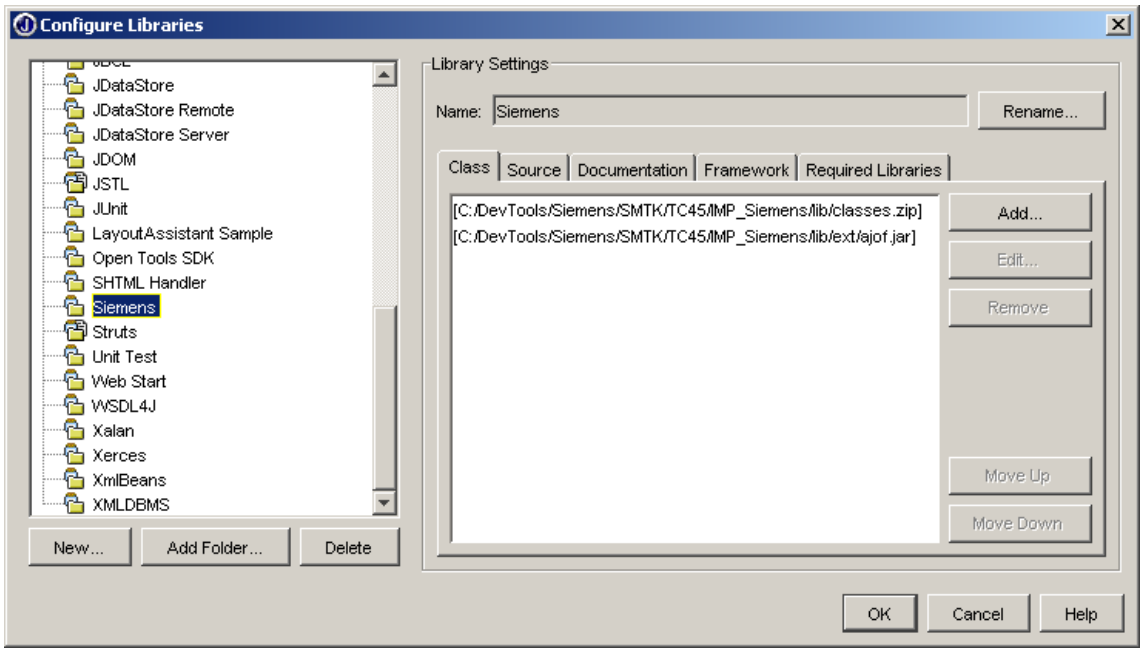


Figure 24: JBuilder - Siemens Library

10.2.2.3 Examples

There are sample projects provided with the SMTK. These projects can be found in the JBuilderSamples directory of the Siemens-TC45-SMTK installation directory. This directory is accessed by opening a project, File->Open Project.

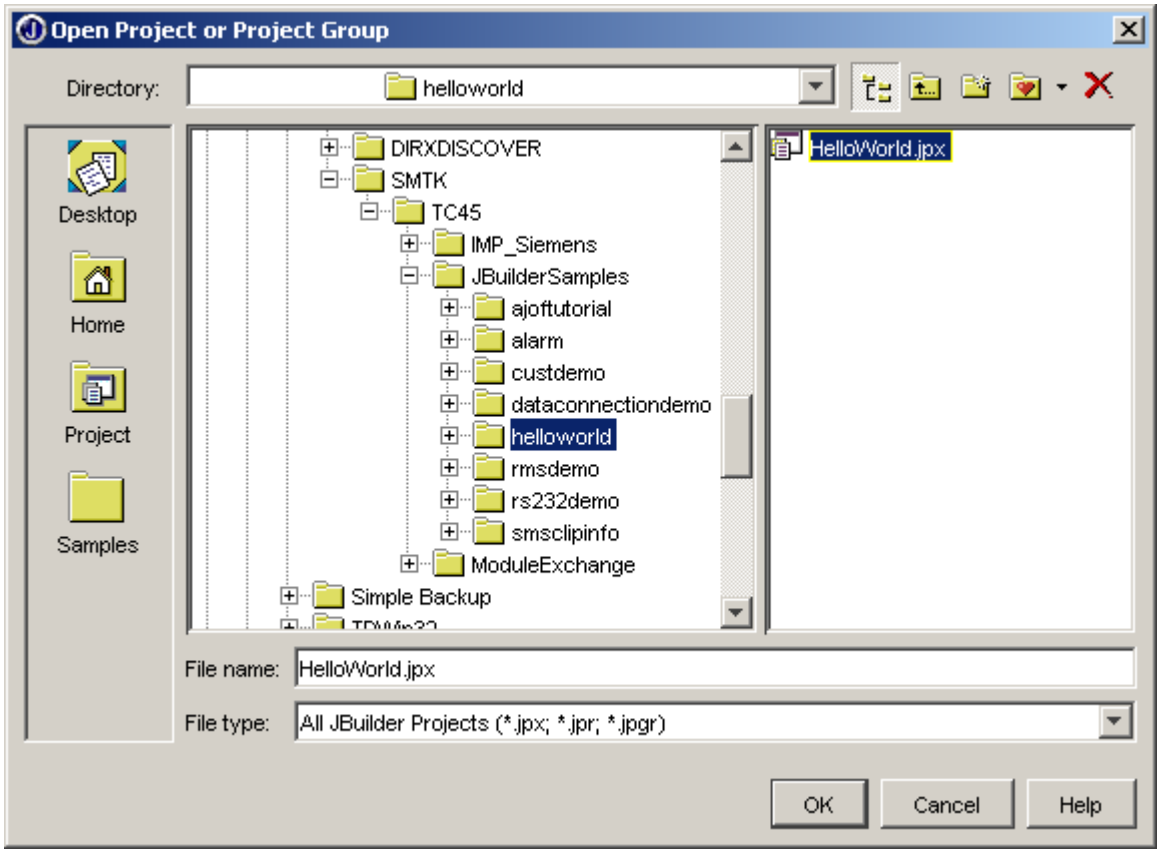


Figure 25: JBuilder - Sample Projects

10.2.2.4 Projects and templates

To create a new project select File->New Project… The New Project Wizard will be launched. An existing project can be used as a template for new project. This means that the initial default values for the new project will be the same as the project chosen as the template. If no project is chosen as a template, the second step of the New Project Wizard gives you the opportunity to choose the emulator used with the project.

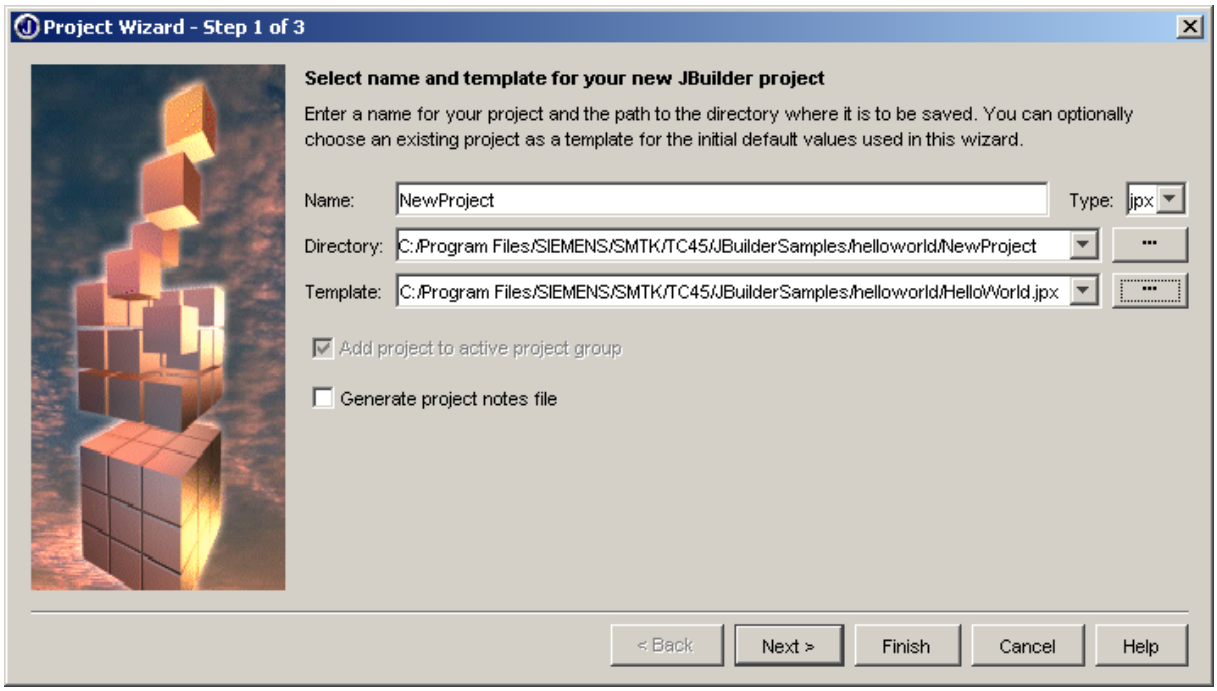

Project Wizard - Step 1 of 3
Select name and template for your new JBuilder project
Enter a name for your project and the path to the directory where it is to be saved. You can optionally choose an existing project as a template for the initial default values used in this wizard.
Name:
NewProject
Type:
jpx
Directory:
C:/Program Files/SIEMENS/SMTK/TC45/JBuilderSamples/helloworld/NewProject
Template:
C:/Program Files/SIEMENS/SMTK/TC45/JBuilderSamples/helloworld/HelloWorld.jpx
Add project to active project group
Generate project notes file
< Back
Next >
Finish
Cancel
Help

Figure 26: JBuilder - Configuring a new project.

You can examine a project's properties by choosing *Project Properties* from the *Project* menu.

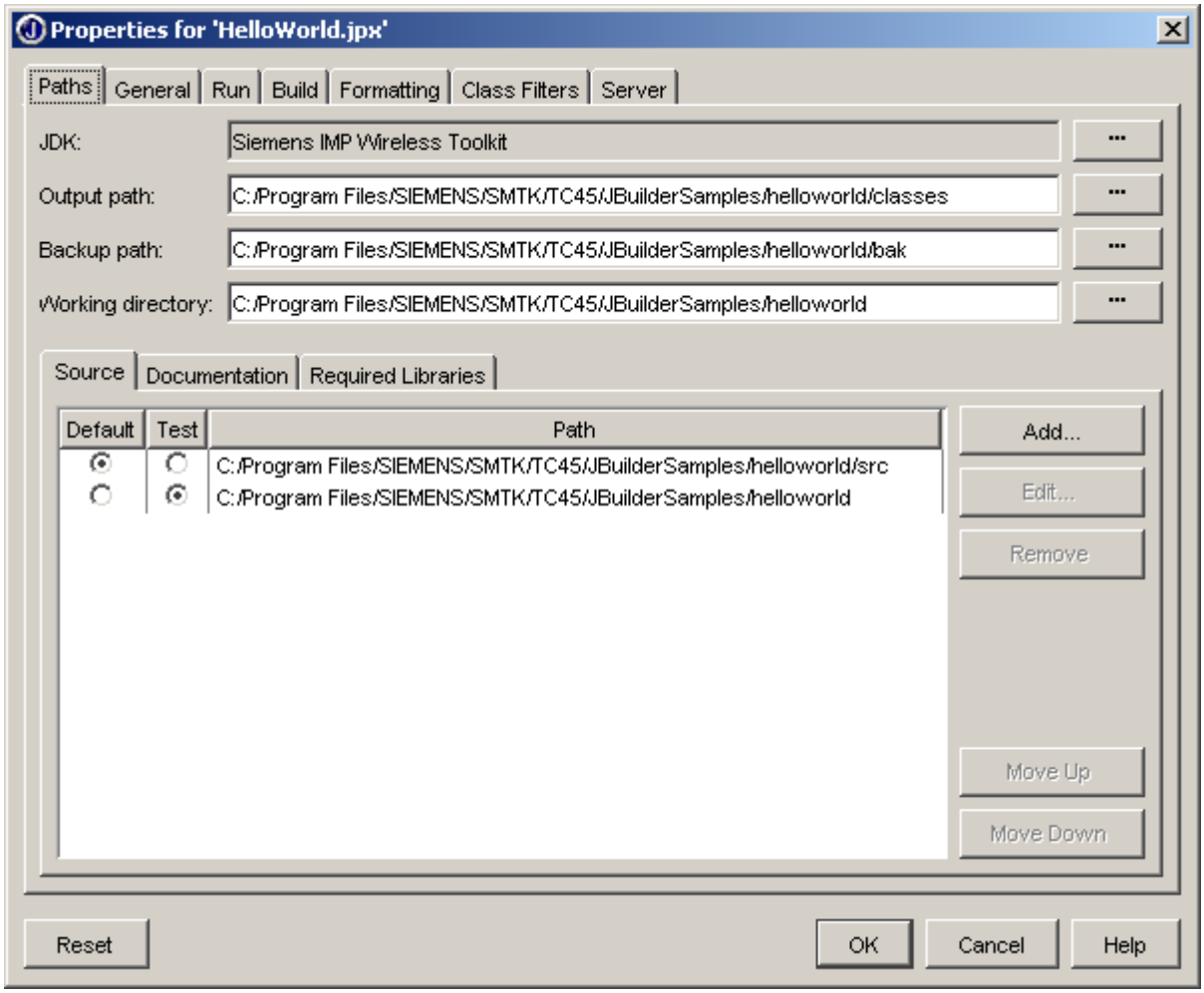


Figure 27: JBuilder - Project Properties

10.2.2.5 Compile and run

Ensure that the proper emulator will be used and compile the project as normal. Any output will be shown in the output window in the IDE.

10.3 Breakpoints

Breakpoints can be set as usual within the IDE. The debugger cannot step through methods or functions whose source code is not available.

11 Java Tutorial

This small tutorial includes explanations on how to use the AT Command API, suggestions for programming MIDlets and an example of using AJOF. The developer should read about MIDlets, Threads and AT commands as a complement to this tutorial.

11.1 Using the AT command API

Perhaps the most important API for the developer is the AT command API. This is the API that lets the developer issue commands to control the module. This API consists of the *ATCommand* class and the *ATCommandListener* and *ATCommandResponseListener* interfaces. Their javadocs can be found in *…\IMPSiemens\doc\html\index.html*, [6].

11.1.1 Class ATCommand

The *ATCommand* class supports the execution of AT commands in much the same way as they would be executed over a serial interface. It provides a simple way to send strings directly to the device's AT parsers.

11.1.1.1 Instantiation with or without CSD support

There can be only exactly as many *ATCommand* instances as there are parsers on the device. If there are no more parsers available, the *ATCommand* constructor will throw *ATCommandFailedException*. A parser with CSD support may be requested through the constructor. As is mentioned in Section 4.3, only one of the parsers supports CSD. If there are no parsers with CSD support available when requested, the constructor will again throw *ATCommandFailedException*. However, an instance may be created with CSD support even if it was not requested.

```
try {
  ATCommand atc = new ATCommand(false);
  /* An instance of ATCommand is created. CSD is not explicitly
   * requested. */
} catch (ATCommandFailedException e) {
  System.out.println(e);
}
```

The csdSupported() method returns the CSD capability of the connected instance of the device's AT parser.

```
boolean csd_support = atc.csdSupported();
```

release() releases the resources held by the instance of the ATCommand class. After calling this function the class instance cannot be used any more but the resources are free to be used by a new instance

11.1.1.2 Sending an AT command to the device, the *send()* method

An AT command is sent to the device by using the send() method. The AT command is sent as a string which must include the finalizing line feed "\r" or the corresponding line end character.

```
String response = atc.send("at+cpin?\r");
/* method returns when the module returns a response */
System.out.println(response);
```

Possible response printed to System.out:
`+CPIN: READY OK`

This send function is a blocking call, which means that the calling thread will be interrupted until the module returns a response. The function returns the response, the result code of the AT command, as a string.

Occasionally it may be infeasible to wait for an AT command that requires some time to be processed, such as *at+cops?*. There is a second, non-blocking, send function which takes a second parameter in addition to the AT command. This second parameter is a callback instance, *ATCommandResponseListener*. Any response to the AT command is delivered to the callback instance when it becomes available. The method itself returns immediately. The *ATCommandResponseListener* and the non-blocking send method are described in Section 11.1.2.

Note: Using the send methods with strings with incorrect AT command syntax will cause mistakes.

11.1.1.3 Data connections

If a data connection is created with the *ATCommand* class, for instance with *'atd'*, an input stream is opened to receive the data from the connection. Similarly, an output stream can be opened to send data on the connection.

```
/* Please note that this example would not work unless the module had
 * been initialized and logged into a network. */

System.out.println("Dialing: ATD" + CALLED_NO);
response = atc.send("ATD" + CALLED_NO + "\r");
System.out.println("received: " + response);

if (response.indexOf("CONNECT") >= 0) {
  try {
    // We have a data connection, now we do some streaming...
    // IOException will be thrown if any of the Stream methods fail
    OutputStream dataOut = ATCmd.getDataOutputStream();
    InputStream dataIn = ATCmd.getDataInputStream();

    // out streaming...
    dataOut.write(new String("\n\rHello world\n\r").getBytes());
    dataOut.write(new String("\n\rThis data was sent by a Java " +
                             "MIDlet!\n\r").getBytes());
    dataOut.write(new String("Press 'Q' to close the " +
                             "connection\n\r").getBytes());

    // ...and in streaming
    System.out.println("Waiting for incoming data, currently " +
                       dataIn.available() + " bytes in buffer.");
```

```
    rcv = 0;
    while(((char)rcv != 'q') && ((char)rcv != 'Q') && (rcv != -1)){
      rcv = dataIn.read();
      if (rcv >= 0) {
        System.out.print((char)rcv);
      }
    }

    /* The example continues after the next block of text */
```

In *…/Siemens/SMTK/TC45/IMPSiemens/src/examples* a complete data connection example, DataConnectionDemo.java, can be found.

These streams behave slightly differently than regular data streams. The streams are not closed by using the close() method. A stream remains open until the *release()* method is called. A module can be switched from the data mode to the AT command mode by calling the *breakConnection()* method.

```
    /* continue example */

    if (rcv != -1) {
      // Now break the data connection
      System.out.println("\n\n\rBreaking connection");
      try {
        strRcv = ATCmd.breakConnection();
      } catch(Exception e) {
        System.out.println(e);
      }
      System.out.println("received: " + strRcv);
    } else {
      // Received EOF, somebody else broke the connection
      System.out.println("\n\n\rSomebody else switched to " +
                         "command mode!");
    }
    System.out.println("Hanging up");
    strRcv = ATCmd.send("ATH\r");
    System.out.println("received: " + strRcv);
  } catch(IOException e) {
    System.out.println(e);
  }
} else {
  System.out.println("No data connection established,");
}
```

An IOException is thrown if any function of the I/O streams are called when the module is in AT command mode

11.1.1.4 Synchronization

For performance reasons there is no synchronization done in this class. If an instance of this class has to be accessed from different threads it has to be ensured that the *send()* functions, the *release()* function, the *cancelCommand()* function and the *breakConnection()* function are synchronized in the user implementation.

11.1.2 ATCommandResponseListener interface

The *ATCommandResponseListener* interface defines the capabilities for receiving the response to an AT command sent to one of the module's AT parsers. When the user wants to use the non blocking version of the *ATCommand.send* function an implementation class for the *ATCommandResponseListener* interface must be created first. The single method of this class, *ATResponse()*, must contain the processing code for the possible response to the sent AT command.

```
class MyListener implements ATCommandResponseListener {

  String listen_for;

  void myListener(String awaited_response) {
    listen_for = awaited_response;
  }

  void ATResponse(String Response) {
    if (Response.indexOf(listen_for) >= 0) {
      System.out.println("received: " + strRcv);
    }
  }
}
```

11.1.2.1 Non-blocking *ATCommand.send()* method

After creating an instance of the *ATCommandResponseListener* class, the class instance can be passed as the second parameter of the non-blocking ATCommand.send() method. After the AT command has been passed to the AT parser, the function returns immediately and the response to the AT command is passed to this callback class later when it becomes available

Somewhere in the application:

```
MyListener connect_list = new MyListener("CONNECT");
atc.send("ATD" + CALLED_NO + "\r", connect_list);

/* Application continues while the AT command is processed*/
/* When the module delivers its response to the AT command the callback
 * method ATResponse is called. If the response is "CONNECT", we will see
 * the printed message from ATResponse in MyListener. */
```

A running AT command sent with the non-blocking send function can be cancelled with *ATCommand.cancelCommand()*. Any possible responses to the cancellation are sent to the waiting callback instance.

Note: Using the send methods with strings with incorrect AT command syntax will cause mistakes.

11.1.3 ATCommandListener interface

The *ATCommandListener* interface implements callback functions for URCs, and changes of the serial interface signals RING, DCD and DSR. The user must create an implementation class for *ATCommandListener* to receive AT events. The *ATEvent* method of this class must contain the processing code for the different AT-Events (URCs) and the *RINGChanged*, *DCDChanged* and *DSRChanged* methods possible processing code for the signal state changes.

11.1.3.1 ATEvents

An ATEvent or a URC is a report message sent from the module to the application. An unsolicited result code can either be delivered automatically when an event occurs or as a result of a query the module received before. However, a URC is not issued as a *direct* response to an executed AT command. Some URCs must be activated with an AT command.

Typical URCs may be information about incoming calls, received SM, changing temperature, status of the battery etc. A summary of URCs is listed in the AT Command Set document [4].

11.1.3.2 Implementation

```
class ATListenerA implements ATCommandListener {

public void ATEvent(String Event) {
  if (Event.indexOf("+CALA: Reminder 1") >= 0) {
    /* take desired action after receiving the reminder */
  } else if (Event.indexOf("+CALA: Reminder 2") >= 0) {
    /* take desired action after receiving the reminder */
  } else if (Event.indexOf("+CALA: Reminder 3") >= 0) {
    /* take desired action after receiving the reminder */
  }

  /* No action taken for these events */
  public void RINGChanged(boolean SignalState) {}
  public void DCDChanged(boolean SignalState) {}
  public void DSRChanged(boolean SignalState) {}
}

class ATListenerB implements ATCommandListener {

  public void ATEvent(String Event) {
    if (Event.indexOf("+SCKS: 0") >= 0) {
      System.out.println("SIM Card is not inserted.");
      /* perform other actions */
    } else if (Event.indexOf("+SCKS: 1") >= 0) {
      System.out.println("SIM Card is inserted.");
     /* perform other actions */
    }
  }

  public void RINGChanged(boolean SignalState) {
    /* take some action when the RING signal changes if you want to */
  }

  public void DCDChanged(boolean SignalState) {
    /* take some action when the DCD signal changes if you want to */
  }

  public void DSRChanged(boolean SignalState {}
    /* take some action when the DSR signal changes if you want to */
  }
}
```

11.1.3.3 Registering a listener with an *ATCommand* instance

After creating an instance of the *ATCommandListener* class, this class instance has to be passed as a parameter to the *ATCommand.addListener()* method. After that, the callback methods will be called by the runtime system each time the corresponding events (URCs or signal state changes) occur on the corresponding device AT parser.

```
/* we have two ATCommands instances, atc1 and atc2 */
ATListenerA reminder_listener = new ATListenerA();
ATListenerB card_listener = new ATListenerB();

atc1.addListener(reminder_listener);
atc2.addListener(card_listener);
```

The *ATCommand.removeListener()* method removes a listener object that has been previously added to the internal list table of listener objects. After it has been removed from the list it will not be called when URCs occur. If it was not previously registered the list remains unchanged.

The same *ATCommandListener* may be added to several *ATCommand* instances and several *ATCommandListeners* may be added to the same *ATCommand*.

11.2 Programming the MIDlet

The life cycle and structure of MIDlets are described in Chapter 5. Since the MIDlets will run on J2ME™, all of J2ME™'s features, including threads, are available. Small applications, such as those without any timer functions or those used only for tests and simple examples, can be written without using threads. Longer applications should be implemented with threads.

11.2.1 Threads

Although small applications can be written without using threads longer applications should use them. The Java programming language is naturally multi-threaded which can make a substantial difference in the performance of your application. Therefore we recommend referring to Java descriptions on threads before making any choices about threading models. Threads can be created in two ways. A class can be a subclass of *Thread* or it can implement *Runnable*.

For example, threads can be launched in *startApp()* and destroyed in *destroyApp()*. Note that destroying Java threads can be tricky. It is recommended that the developer read the Java documentation on threads. It may be necessary to poll a variable within the thread to see if it is still alive.

11.2.2 Example

/* This example derives a class from Thread and creates two instances
 * of the subclass. One thread instance finishes itself, the other one
 * is stopped by the main application. */

```
package example.threaddemo;

import javax.microedition.midlet.*;

public class ThreadDemo extends MIDlet {

  /* Member variables */
  boolean    runThreads = true; // Flag for stopping threads
  DemoThread thread1;           // First instance of DemoThread
  DemoThread thread2;           // Second instance of DemoThread

  /* Private class implementing the thread to be started by the
   * main application */
  private class DemoThread extends Thread {
    int loops;

    public DemoThread(int waitTime) {
      /* Store number of loops to execute */
      loops = waitTime;
      System.out.println("Thread(" + loops + "): Created");
    }

    public void run() {
      System.out.println("Thread(" + loops + "): Started");
      for (int i = 1; i <= loops; i++) {
        /* Check if main application asked thread to die */
        if (runThreads != true) {
          System.out.println("Thread(" + loops + "): Stopped from outside");
          /* Leave thread */
          return;
        }
        /* Print loop counter and wait 1 second,
         * do something useful here instead */
```

```
          System.out.println("Thread(" + loops + "): Loop " + i);
          try {
            Thread.sleep(1000);
          } catch(InterruptedException e) {
            System.out.println(e);
          }
        }
        System.out.println("Thread(" + loops + "): Finished naturally");
      }
    }

    /**
     * ThreadDemo - constructor
     */
    public ThreadDemo() {
      System.out.println("ThreadDemo: Constructor, creating threads");
      thread1 = new DemoThread(2);
      thread2 = new DemoThread(6);
    }

    /**
     * startApp()
     */
    public void startApp() throws MIDletStateChangeException {
      System.out.println("ThreadDemo: startApp, starting threads");
      thread1.start();
      thread2.start();
      System.out.println("ThreadDemo: Waiting 4 seconds before stopping threads");
      try {
        Thread.sleep(4000);
      } catch(InterruptedException e) {
        System.out.println(e);
      }
      destroyApp(true);
      System.out.println("ThreadDemo: Closing application");
      notifyDestroyed();
    }

    /**
     * pauseApp()
     */
    public void pauseApp() {
      System.out.println("ThreadDemo: pauseApp()");
    }

    /**
     * destroyApp()
     */
    public void destroyApp(boolean cond) {
      System.out.println("ThreadDemo: destroyApp(" + cond + ")");
      System.out.println("ThreadDemo: Stopping threads from outsdide");
      runThreads = false;
      try {
        System.out.println("ThreadDemo: Waiting for threads to die");
        thread1.join();
        thread2.join();
      } catch(InterruptedException e) {
        System.out.println(e);
      }
      System.out.println("ThreadDemo: All threads died");
    }
}
```

11.3 AJOF

The following example uses AJOF. It demonstrates how the writing of an application for the module is abstracted and simplified with the framework. This example does not use threads but could be implemented in a structure similar to the one shown above.

Some comments:

- *CWmMIDlet* is the MIDlet instance wrapping the application. *CWmMIDlet* adds AT command channel management to the standard MIDlet class.
- An AT channel, *ATChannel*, consists of an ATC parser and one or more listeners waiting for URCs. The channel is distributed by the application to any class that needs access to the module via the ATC interface.

11.3.1 Example

```
/* WmTutorial.java
 * Copyright (C) Siemens AG 2003. All Rights reserved.
 * Transmittal, reproduction and/or dissemination of this document
 * as well as utilization of its contents and communication thereof
 * to others without express authorization are prohibited.
 * Offenders will be held liable for payment of damages.
 * All rights created by patent grant or registration of a utility
 * model or design patent are reserved.
 */

package example.ajoftutorial;

import com.siemens.icm.io.ATCommandFailedException;
import com.siemens.icm.ajof.WmMIDlet;
import com.siemens.icm.ajof.AtChannel;
import com.siemens.icm.ajof.AtChannel.SendTimeoutException;
import com.siemens.icm.ajof.AjofException;
import com.siemens.icm.ajof.status.WmLock;
import com.siemens.icm.ajof.status.WmLockException;
import com.siemens.icm.ajof.phonebook.SimplePhonebook;
import com.siemens.icm.ajof.phonebook.PhonebookStoreException;
import com.siemens.icm.ajof.sms.SimpleSms;
import com.siemens.icm.ajof.sms.SmsSendException;

/**
 * demonstrates how easy it is to retrieve a number from the phonebook and send
 * an SMS message to this destination.
 * @author SIEMENS AG
 * @version 1
 * @since AJOF 1.0
 */
public class WmTutorial extends WmMIDlet
{
    // Please, change these constants to your desired settings
    final String RECIPIENT_NAME       = "foo, bar";
    final String SERVICE_CENTRE_ADDR = "+491720000000";
    final String SIM_PIN              = "0000";
    final String MESSAGE_TXT          = "Hello, world.";

    /**
     * creates and sends an SMS message to a recipient, whose phone number is
     * retrieved from the SIM phonebook.
     * <ol><li>get an AT channel to communicate with the module's AT interface
     * <li>just check, if the module answers to AT commands,
     * <li>enter the <code>SIM PIN</code>, so we can access the phonebook and
     * send an SMS message,
     * <li>create a phonebook control and look up the recipient's phone number,
     * <li>create an SMS control,
     * <li>set the service centre address (SCA),
```

```
     * <li>last, but not least send the SMS message.
     * <li>Please do not forget to cleanup all used resources. Each AJOF class has
     * a method <code>release()</code>, which performs this task.
     * <li>Terminate the application by calling <code>notifyDestroyed()</code>.</ol>
     * @see <code>AT</code>
     * @see <code>AT+CPIN</code>
     * @see <code>AT+CPBR</code>
     * @see <code>AT+CPMGF</code>
     * @see <code>AT+CPMGS</code>
     */
    public void startApp()
    {
        try
        {
            // To access the module we need an AT channel:
            AtChannel atch = openAtChannel();

            // Test the connection
            // (this code is optional and used only for demonstration purposes)
            System.out.println("\nTesting connection...");

            String response = atch.send("at\r");
            System.out.println("atC + response);

            // Set elaborated error messages
            atch.send ("AT+CMEE=2\r");

            // Enter the SIM PIN
            System.out.println("Entering PIN...");
            WmLock lockControl = new WmLock(atch);

            try
            {
                lockControl.sendSimPin(SIM_PIN);

                // We create a CSimplePhonebook instance
                // to look into the module's phonebook.
                SimplePhonebook pbk = new SimplePhonebook(atch);

                try
                {
                    // Now let's look for the SMS's recipient.
                    // The number is stored in the phonebook.
                    System.out.println("Looking up phone number for " +
                                       RECIPIENT_NAME + ", please wait...");
                    String destinationAddr = pbk.getNumber(RECIPIENT_NAME);

                    // create a simple SMS control
                    SimpleSms sms = new SimpleSms(atch);

                    try
                    {
                        // You need to set the SCA only once in your application
                        sms.setServiceCentreAddr(SERVICE_CENTRE_ADDR);

                        // now send the message
                        System.out.println("Sending Short Message, please wait...");
                        sms.send(destinationAddr, MESSAGE_TXT);
                        System.out.println("Short Message was sent.");
                    }

                    catch (SmsSendException e)
                    {
                        System.out.println(e);
                    }

                    finally
                    {
                        // release the used resources
                        sms.release();
                        sms = null;
                    }
                }
```

```
                catch (PhonebookStoreException e)
                {
                    System.out.println(e);
                }

                finally
                {
                    // release the used resources
                    pbk.release();
                    pbk = null;
                }
            }

            catch (WmLockException e)
            {
                System.out.println(e);
            }

            finally
            {
                // release the used resources
                lockControl.release();
                lockControl = null;
            }
        }

        catch (ATCommandFailedException e)
        {
            System.out.println(e);
        }

        catch (SendTimeoutException e)
        {
            System.out.println("AT_Command timeout");
        }

        finally
        {
            // Terminate the application.
            // You don't need to care about closing open AT channels.
            // CWmMIDlet does this for you.
            System.out.println("Now terminating application.");
            destroyApp(true);
        }
    }

    /**
     * Your application does not necessarily need to override
     * <code>destroyApp()</code>, but if it does, make sure that
     * <code>super.destroyApp()</code> is called, to release the AT channels.
     */
    public void destroyApp(boolean unconditional)
    {
        // your finalization code goes here...

        super.destroyApp(unconditional);
        notifyDestroyed();
    }
}
```